IN THE UNITED STATES DISTRICT COURT
FOR THE SOUTHERN DISTRICT OF NEW YORK

INTERNATIONAL BUSINESS MACHINES CORPORATION,

Plaintiff,

-vs.-

PLATFORM SOLUTIONS, INC., and T3 TECHNOLOGIES, INC.,

Defendants.

DECLARATION OF DR. MARK SMOTHERMAN IN SUPPORT OF IBM'S CLAIM CONSTRUCTION REPLY BRIEF FOR THE *MARKMAN* HEARING

Civil Action No. 06 CV 13565 (LAK)

DECLARATION OF DR. MARK SMOTHERMAN

I, Dr. Mark Smotherman, declare as follows:

Architecture and Implementation

1. I believe that much of the disagreement over the terms at issue in this case involves the difference between the architecture of a computer system and the physical implementation(s) of that architecture.

2. It has been well known since the 1960's through today that different implementations of a particular instruction set architecture can use different design techniques based on different price performance goals to provide that same architecture. This idea was introduced by Amdahl, Blaauw, and Brooks in the 1960's and quickly became and remains to this day a fundamental concept of computer design:

> It required a new concept and mode of thought to make the compatibility objective even conceivable. In the last few years, many computer architects had realized, usually implicitly, that logical structure (as seen by the programmer) and physical structure (as seen by the engineer) are quite different. Thus each may see registers, counters, etc., that to the other are not at all real entities. This was not so in the computers of the 1950's. The *explicit* recognition of the duality of structure opened the way for compatibility within System/360. The compatibility requirement dictated that the basic architecture had to embrace different technologies, different storage-circuit speed ratios, different data path widths, and different data-flow complexities.

G.M. Amdahl, *et al.*, Architecture of the IBM System/360, *IBM J. Res. Develop.*, Vol. 8, No. 2 at 89-90 (1964), available on-line as http://www.research.ibm.com/journal/rd/082/ibmrd0802C.pdf (emphasis in original, underline added).

3. Furthermore, a particular instruction set architecture is typically re-implemented over time so that existing programs do not have to be repeatedly re-written for newer, faster, and more reliable technology. Thus an instruction set architecture will typically last for decades,

albeit with some extensions, while implementations of that architecture typically last only for a few years.

4. It is therefore inappropriate to restrict architectural concepts to the particular design choices of one implementation or a subset of possible implementations.

Processor

5. I agree with Dr. Patt that microcode is a part of some processors. However, I disagree with Dr. Patt's contention that microcode is not a form of software. (Patt Decl. ¶ 7.) Microcode is in fact a form of software that controls the interpretation and execution of instructions by a processor.

6. I also disagree with Dr. Patt's view that technological changes, including the increase in the number of transistors available to a computer designer, had by the 1990's somehow so narrowly constricted the design choices available to a computer designer that there was only one way for microcode to be used as part of a processor. (Patt Decl. ¶ 9.) I further believe that Dr. Patt is inconsistent in his opinions that (a) microcode can be stored in RAM (Patt Decl. ¶ 8) but that (b) somehow microcode could only be "burned into a chip" during the 1990's and therefore could be neither loaded nor modified (Patt Decl. ¶ 7). By definition, the contents of a RAM can be loaded and modified.

7. While it is true that some processors have been built with microcode in an unchangeable on-chip ROM, it is equally true that processors have been built with microcode being loaded and stored in RAM. This latter approach uses what is called writeable control store (WCS) or microcode RAM, in which microcode is not burned into a processor but rather is loaded (e.g., from disk) when the processor is booted or even dynamically loaded (i.e., modified) while the computer system is running. It is also well known in the art to have a combination of

both microcode ROM and microcode RAM, with some processors providing the opportunity for microcode loaded into the RAM to override (i.e., patch) errors in the ROM-based microcode. This patching can occur even after the processor is delivered to a customer. I believe that dynamic loading of microcode and patching of microcode are clear instances of "modifying" microcode.

8. Two examples of processors from the 1990's with microcode that could be dynamically loaded while the computer system is running are the IBM AS/400 and the Texas Instruments VCP. The IBM AS/400 is described in this manner: "The processor fetches and executes Internal Microprogrammed Interface (IMPI) instructions. As the name implies, each IMPI instruction is actually executed as a microprogram. The microcode is called Horizontal Licensed Internal Code (HLIC). ...The control store array contains the most commonly executed microcode words. Additional microcode is loaded from memory in an overlay area as required during execution." Q.G. Schmierer and A.H. Wottreng, IBM AS/400 processor architecture and design methodology, *Proc. ICCD* (1991) at 440 (Ex. A). The Texas Instruments VCP is a vector coprocessor: "The VCP is a microprogrammable, pipelined processor containing multiple arithmetic resources, as shown in the block diagram in Figure 3. All resources are controlled by an on-chip microsequencer executing from a 256 instruction by 192-bit, on-chip microcode RAM. ...The CPL controller handles loading and executing of microcode routines as directed by Command Parameter Lists implemented as application data structures loaded into the local BPM memory." R. Branstetter, *et al.*, Ultra-reliable digital avionics (URDA) processor architecture, *Proc. NAECON* (1994) at 276 (citations omitted) (Ex. B).

9. An example of loading microcode from a disk is found in an IBM patent, US 5,371,848, "Method for operating and displaying status of a computer system," filed in April

1993 and issued in December 1994: "Next, the activate microcode 23 reads the profile data to learn the IML or power-on reset parameters, which indicate the location of the microprogram that is to be loaded into memory 15 of CEC 12, how much expanded and central store memory is required, a channel configuration, and a name of the microprogram to be loaded (step 114). After reading these parameters, the activate microcode 28 directs the loading of microcode from disk 24 to memory 15 (steps 114-116)." ('848 patent, col. 4:6-15, underlines added.)

10. An example of a processor with a microcode ROM along with a microcode RAM used for microcode patching was described by DEC: "The microsequencer and EBOX work together as a 4-stage micropipeline to execute instructions under microprogrammed control. The control store is a 1600 61b microword ROM. The microcode can be patched using a CAM to substitute RAM words for ROM." R. Badeau, *et al.*, A 100 Mhz macropipelined CISC CMOS microprocessor, *Proc. ISSCC* (1992) at 104 (Ex. C). As another example of patching, Intel designed the Pentium Pro and the Pentium II processors with a microcode update facility. *See* Chris Oakes, Intel Studies End-User Processor Patches, *Wired News*, July 9, 1997, available on-line as http://www.wired.com/science/discoveries/news/1997/07/5062.

11. An example of microcode patching as well as loading microcode from a disk is discussed in an Intel patent, US 6,154,834, "Detachable processor module containing external microcode expansion memory," filed in May 1997. This patent includes the following analysis of the state of microcode technology in the 1990's:

> As shown in FIG. 2, for computers, the microprocessor 200 is usually attached directly to a motherboard 210 through a multi-pin socket 220. In this embodiment, microprocessor 200 includes core memory 230 capable of containing microcode. A fixed portion of this core memory 230 is writable during execution of Basic Input/Output System (BIOS) code contained in on-substrate, non-volatile memory 240. The storage space of this non-volatile memory 240 (referred to as a BIOS memory space) is also fixed in size.

> Over the last few years, some disadvantages concerning the loading of microcode patches have been uncovered.
> ...
> Another disadvantage is that microcode patches currently are distributed by diskette or through an electronic bulletin board on the Internet. These microcode patches are subsequently loaded into memory 230 of FIG. 2.

"Description of Related Art" ('834 patent, col. 1:40-51, 2:7-10; underlines added). The use of "core" here refers to the processor core.

12. Examples of processors in the 1980's and 1990's that provide for microcode corrections and updates applied after the processors have been delivered to customers include the IBM ES/3090 and the IBM ES/9000 Model 982. The IBM ES/3090 used a writeable control store to provide this facility: "The ES/3090 is a horizontal microcode controlled processor ... A key feature of the writeable control storage is that this allows ES/3090 to participate in future architecture enhancements without requiring hardware changes in the field." W.J. Nohilly, ES/3090: A realization of ESA/370 system architecture in IBM's most powerful mainframe computer through a balance of technology and system innovations, *Proc. ICCD* (1988) at 611-612 (Ex. D). The IBM ES/9000 Model 982 also provided for microcode updates: "Service mode is also used to apply updates to the microcode (formally called licensed internal code). The PCEs, IOPs, channels, and CPUs all have microcode. Service personnel usually can change the CPC's current active microcode level without any downtime. Microcode changes can be downloaded via TP link to the PCE disk at any time. The PCE is placed in service mode, and the backup side disk merges the changes with the current code to form the new microcode level. The new level is then written to the active side and the appropriate control storages within the CPC." L. Spainhower, *et al.*, IBM's ES/9000 Model 892's fault-tolerant design for consolidation, *IEEE Micro*, Vol. 14, No. 1 (1994) at 58 (Ex. E).

Instructions and Instruction Sets

13. I disagree with Dr. Patt in his assertion that assembly language instructions are "statements" and not "instructions." (Patt Decl. ¶ 14.) In his textbook, Dr. Patt writes: "Instead of an instruction being 16 0s and 1s, as in the case of the LC-2 ISA, an instruction in assembly language consists of four parts, as shown below: LABEL OPCODE OPERANDS ; COMMENTS" Y.N. Patt and S.J. Patel, *Introduction to Computing Systems*, McGraw-Hill, (2001), at 141 (underlines added) (Ex. F). Dr. Patt is clearly differentiating an assembly language instruction from a machine instruction, which is a string of digits created by translation of assembly language instructions. Furthermore, in his textbook, there are numerous references to assembly language instructions using the unqualified term "instruction," including this description of the assembly language instruction **LD R2, NUMBER**: "The LD instruction (line 06) requires two operands (the memory location from which the values is to be read) and the destination register which is to contain the value after the instruction completes execution." *op cit.* at 142. Dr. Patt also gives assembly language instructions for implementing a function call and describes this code as being "executed":

> The LC-2 code to perform this function call looks as follows:
> **LDR R0, R6, #3 ; load a**
> **STR R0, R6, #8 ; store the first argument to NoName**
> ...
> The first seven LC-2 instructions accomplish the task of transmitting the argument values.
> ...
> Figure 14.6 shows the layout in memory of these two activation records after this code has executed.

op cit. at 323 (underlines added). This usage of instruction and execution with relation to assembly language instructions is readily understood but contradicts Dr. Patt's opinion expressed in his declaration that it is incorrect to use the idea of "executing" with relation to assembly language statements. (Patt Decl. ¶ 14.) In summary, Dr. Patt's assertion in his declaration that

"the terms 'instruction' and 'machine instruction' are synonymous in the field of computer architecture" (Patt Decl. ¶ 16) is inconsistent with the statements he previously made in his textbook.

14. I agree with Dr. Patt regarding the following statement found in his textbook: "There is no confusing an instruction in a low-level language with a statement in English." Patt and Patel, *op cit.* at 140 (Ex. F).

15. Computer systems can be designed with multiple processors, some of which have one instruction set architecture while others have a different instruction set architecture. One example is a PC system from Myriad that included an x86 processor, executing its instruction set, and an i860 processor, executing a different instruction set. S.C.J. Garth, Combining RISC and CISC in PC systems, *IEE Colloq. On RISC Archs. And Appls.*, 1991 (Ex. G). Another example is a server system from Supermicro that included Pentium Pro processors, executing the Intel x86 instruction set, and an i960 processor, executing a different instruction set. Supermicro's I2O architecture server solution reduces cost of ownership and improves scalability, Oct. 8, 1997, available on-line at http://www.supermicro.com/newsroom/pressreleases/1997/press100897.cfm. Such heterogeneous designs were also standard practice for supercomputers, with one example being the Cray T3D, which used DEC Alpha processors for the processing elements along with an I/O host with a different instruction set, such as a Cray Y-MP. R.K. Koeninger, *et al.*, A shared memory MPP from Cray Research, *Digital Tech. J.*, Vol. 6, No. 2 (1994) available on-line at http://www.hpl.hp.com/hpjournal/dtj/vol6num2/vol6num2art1.pdf.

Registers

16. I disagree with Dr. Patt that the term "register" can only mean a "hardware storage element *in* the processor that (among other qualities) could be accessed much faster than memory." (Patt Decl. ¶ 22, emphasis in original). I believe that this is an instance in which he is attempting to restrict an architectural concept to a particular implementation. It is an implementation decision as to whether a register defined in the architecture should be implemented in a hardware register or in memory. Based on price/performance goals, a computer designer could choose one implementation method or the other. This is an example of the difference between the viewpoint of a programmer and the viewpoint of an engineer as discussed by Amdahl, Blaauw, and Brooks (see paragraph 2 of this declaration).

17. An example of a computer system having architected registers in main memory is the Texas Instruments TMS 9900: "9900 processors contain three primary internal registers: the program counter (PC), the status register, and the workspace pointer (WP). ... The WP points to a starting location in memory of the 16 general-purpose 'workspace' registers, which must be held in contiguous words. A 'context switch' mechanism loads the WP with a new value, thus defining a new set of work-space registers in main memory. ... The 9900 family supports a memory-to-memory architecture, although workspace registers are provided to reduce program size through encoding efficiency." R.V. Orlando and T.L. Anderson, An overview of the 9900 microprocessor family, *IEEE Micro*, Vol. 1, No. 3 (1981) at 38-39 (Ex. H).

18. A second example of architected registers in main memory is US 5,590,345, "Advanced parallel array processor (APAP)," filed in May 1992 and issued in December 1996. The clear distinction between architecture (that is, the interface to the application software) and implementation is evident in the design choice of mapping some registers to memory in one implementation but not doing so in another implementation:

> The major parts of the internal data flow of the processing element are shown in FIG. 7.
> ...
> The important paths of the internal data flows use 12 nanosecond hard registers such as the OP register 450,
> ...
> Other required registers are mapped to memory locations.
> ...
> As circuit size shrinks and greater packaging density becomes possible then data flow elements like base and index registers, currently mapped to memory could be moved to hardware.
> ...
> Very importantly, this hardware alternative does not affect any software.

('345 patent, col. 29:36-30:34; underlines added).

Floating Point Unit

19. I disagree with Dr. Patt that a floating point unit cannot encompass forms of software. (Patt Decl. ¶ 23.) From the 1960's through today, it was well known that a floating point unit could be implemented in many ways. For example, a floating-point unit can be implemented using fixed control circuitry ("hardwired") or using microcode. An example of the latter design approach is the Gmicro/FPU, in which a floating point unit is described as containing its own microcode: "Figure 9 shows the Gmicro/FPU's internal structure. The three main elements are: ... Floating-point execution unit (ECU), which contains the microcode ROM ..." S. Kawasaki, *et al.*, A floating-point VLSI chip for the TRON architecture: An architecture for reliable numerical programming, *IEEE Micro*, Vol. 9, No. 3 (1989) at 36 (Ex. I).

Routine

20. The term sequence is generally taken to mean a list of items that are in one-to-one correspondence with the natural numbers, 1, 2, 3, ... Thus, we can describe items within a sequence as "first," "second," etc., and we can describe the relation between one item in a sequence and another item as the one being "before" or "after" the other item. In a computer

program, a "sequence" of statements or instructions follows this general idea: "Sequencing is the simplest structuring mechanism available in programming languages. It is used to indicate that the execution of a statement (B) must follow the execution of another statement (A)." C. Ghezzi and M. Jazayeri, *Programming Language Concepts*, Wiley and Sons, (1982), at 129 (Ex. J).

21. In advanced processors, such as the Intel Pentium Pro of the mid 1990's, there is a complication in describing the execution ordering of the machine instructions from a routine. In this type of processor, machine instructions can be executed out of their normal order to take advantage of some machine instructions being ready to execute (with all dependencies being satisfied) while others that were fetched prior to the currently-ready machine instructions must wait (the wait being forced because not all dependencies have been satisfied). Moreover, since some machine instructions can be decomposed into groups of micro-operations within these advanced processors, there is still further complexity in describing the execution ordering of these micro-operations, some of which from one machine instruction may be interspersed in execution with others from another machine instruction. In fact, in some advanced processors these groups of decoded micro-operations can be cached and can be executed multiple times without the original machine instructions being re-fetched or re-decoded. Indeed, Dr. Patt worked on the design of just such an advanced processor in the mid-1980's. *See* J.E. Wilson, S.W. Melvin, M.C. Shebanow, W.-M. Hwu, and Y.N. Patt, "On Tuning the Microarchitecture of an HPS Implementation of the VAX," *ACM SIGMICRO Newsletter*, Vol. 19, No. 3 (1988) (Ex. K). Thus, to identify an execution sequence related to a routine, we may need to qualify exactly what level of execution is being described when executing on an advanced processor.

I declare under penalty of perjury that the forgoing is true and correct.

Executed at Clemson SC, June 27, 2008

Mark Smotherman, Ph.D.

IBM AS/400 Processor Architecture and Design Methodology

Quentin G. Schmierer Andrew H. Wottreng
IBM Application Business Systems
Rochester, Minnesota 55901

Abstract

The IBM AS/400 is a family of general purpose mid-range computers specifically designed to run commercial business applications and transaction processing in batch and interactive environments. The April, 1991 announcement is the second generation of the system. It provides a 2.7 times performance [3] improvement over its predecessor. Architectural support for multi-processors is also part of the product. To achieve this performance improvement required significant changes to the processor architecture, chip technologies and operating system hardware. This paper briefly describes the architecture and design methodology used in the IBM AS/400 processor.

The processor chip designs made extensive use of high level hardware description languages and logic synthesis. Each design group was responsible for defining its design in VHDL and converting the design into logic using logic synthesis. The designers' experiences with these new methods are highlighted.

Introduction

The processor architecture design was key to improving the performance of the second generation IBM AS/400 system. Before the project began, it was clear that technology improvements alone would not be enough. The processor architecture had to be greatly enhanced. The major architectural additions were a store-through data/instruction cache, expanded memory addressability, improved floating point performance and the ability to operate in a multiple processor environment. The multi-processor capability had to be implemented almost completely transparent to the operating system. To make the project "fly" required design tool enhancements and increased reliance on logic synthesis. The remainder of this paper briefly describes the processor architecture and the design methodology used to create it.

Architecture

The AS/400 processor was fabricated from an IBM low power, standard cell CMOS technology. There are five 12.7mm logic chips and two 8k x 18 SRAM chips packaged on two 44mm multi-chip modules (MCMs). The cache and control store are individually packaged static RAMs. The processor clock cycle is 45ns worst case. This was the same clock cycle as the first generation processor had using a less dense bipolar technology.

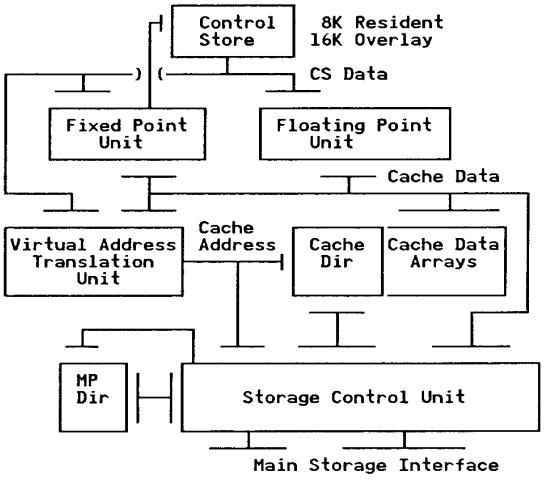


Figure 1 IBM AS/400 Processor

The processor fetches and executes Internal Microprogrammed Interface (IMPI) instructions. As the name implies, each IMPI instruction is actually executed as a microprogram. The microcode is called Horizontal Licensed Internal Code (HLIC). An IMPI instruction may be two to six bytes in length aligned on byte boundaries. The IMPI instruction is executed as a sequence of one or more microcode words. Each microcode word is 60 bits in width with an additional 12 bits used for error correction coding (ECC) and flags. This long width allows several operations to be executed independently during the same processor cycle. The control store array contains the most commonly executed microcode words. Additional microcode is loaded from memory in an overlay area as required during execution. Figure 1 shows the basic components of the AS/400 processor architecture. The main storage interface is time shared with the I/O subsystem.

The fixed point unit executes control words that use fixed point arithmetic and other logical operations. It contains an array of General Purpose Registers (GPRs) accessible only to microcode. A 32-bit arithmetic logic unit (ALU), a halfword decimal ALU and a halfword multiplier are also implemented. The logic required to fetch the next microcode word based on the current processor state is in this unit. The first microcode word for each IMPI instruction is based on the IMPI opcode. A floating point unit, adhering to IEEE standard 754 for binary and floating point arithmetic, is included. It contains eight floating point registers (FPRs) capable of single and double precision arithmetic on addition, subtraction, multiplication, division, comparison and square root.

All references to storage in an IMPI instruction are through virtual addresses. The virtual address translation unit (VAT) converts virtual addresses into real memory addresses. The operating system works with 16-byte virtual addresses, although only 6-byte addresses are directly manipulated in the hardware. Fetches and stores to memory are initiated, as required, by the microcode instructions. The IMPI instructions directly manipulate sixteen 32-bit registers. They may contain addresses or program data. A set of microcode accessible registers called resolved address registers (RAR) store translated virtual addresses for reuse during IMPI instruction execution. IMPI instruction fetching and buffering are done here, also under hardware control.

The storage control unit (SCU) manages memory accesses. Data may be retrieved from either the 128kb data/instruction cache or directly from memory. Each cache line is 32 bytes. The SCU detects cache misses and writes. Since the cache is store-through, all writes are done to both cache and main memory. The SCU detects cache misses and retrieves data from memory, as needed. Its secondary functions are resolving virtual address translation table misses and arbitrating the main store interface usage with the I/O subsystem. Multi-processor contention for the main store interface is also arbitrated here.

The main storage interface consists of a command buss and a 64-bit wide data buss. The I/O subsystem can store and fetch data directly from memory. Up to eight memory cards of 64mb each may be attached to the interface. Other processors may also be connected. When a multi-processor configuration is implemented, they share a single copy of main memory. All interface busses are synchronous with the processor clock using sophisticated clock synchronization logic. Error correction, diagnostics and DRAM refresh are implemented on the memory cards in two 12.7mm CMOS logic chips. Multiple I/O busses are connected to the I/O subsystem. The first buss is wired directly while the remainder are connected using fiber optic cables. Redundancy paths are implemented to provide alternate means for data transfer in case one of the fiber optic channels fails. Each I/O buss is 32 bits in width and asynchronous. Data streaming is supported.

Design Methodology

The design methodology for the AS/400 processor had to deal with significant change. Designers had to describe a new architecture and convert that description into working hardware as efficiently as possible while meeting schedules. We chose, for the first time in a processor design, to use logic synthesis for the transcription process. We selected VHDL as a design description language. Many of the designers had experience with earlier internal IBM register transfer design languages. We had synthesized earlier designs into logic. Success had been mixed. Synthesis worked well as long as performance was not a major issue. The internal design language did not allow enough design control for a high performance application. VHDL, with its mix of functional description constructs and attribute information offered a way of putting more designer input into the synthesis process. This ability proved crucial in achieving the design goals.

Figure 2 shows the general flow of our design process. This paper will not cover all steps in the process, but only those that differed from previous projects. The first step in the process was documentation. Because of the massive amount of change involved, documentation was very important. Without good documentation at the start, it is very easy for subtle errors to creep in and escape detection until late in the schedule. We spent much time preparing a formal architectural specification. This was reviewed by each design team and used as a starting point and reference for each design. The microcode format was also defined at this time.

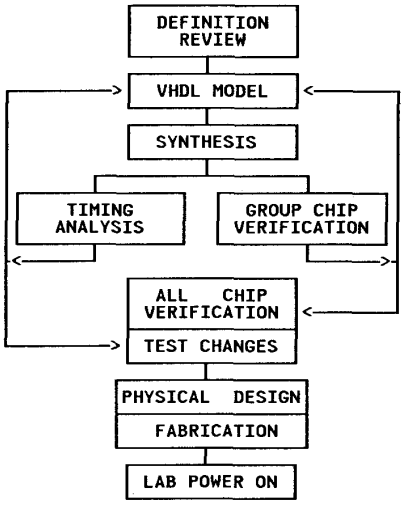


Figure 2 AS/400 Design Process

The chip design teams began the implementation phase by defining chip partitions. The processor was targeted to fit into five CMOS chips. Once the function of each chip was roughly defined, an I/O specification document was created. It contained a detailed description of each chip level net and how each chip connected to it. The design teams reviewed the document periodically and worked out differences and "fuzzy" definitions. A program was used to screen the document for changes and alert the proper chip teams. All changes had to be reviewed and approved. This reduced errors and misinterpretations significantly and led to a cleaner first implementation. The chip interfaces were defined as VHDL entities. Each chip was internally subdivided into "chiplets". Chiplet interfaces were then defined as VHDL entities.

In addition to the formal documents, each chip team attempted to document their design as they wrote their VHDL. In some cases this was in the form of a separate chip workbook. In most cases the workbook was simply a collection of profusely commented VHDL models. The comments were written first to describe the logic function; the actual VHDL defining the function was added later. The VHDL functional description was not, by itself, sufficiently clear to document the design for review purposes. More clarity was sacrificed as the design was modified later in the project to resolve timing problems.

Logic Entry

The chip logic designs were described in VHDL. To allow reuse of existing synthesis and simulation tools, some compromises were required. The major compromise was to use a limited subset of the language. Only the concurrent set of VHDL constructs, excluding process statements, was used. The subset correlated closely in function to previously used register transfer language descriptions. This shortened the designers' learning curve on the language as well. VHDL attribute information was used heavily to provide synthesis directives. The designer was able to give synthesis either suggestions or explicit directives on how to form logic. This made up for weaknesses in the synthesis algorithms and gave the designer pinpoint control of results. Control of results was extremely important in achieving a high performance design. Synthesis produces good implementations in 95 percent of the design. But finish the last five percent requires precise designer input.

Standard text editors were used to enter the VHDL. There were cases where graphical entry might have been useful, but for the majority of the logic, text entry was sufficient. Textual comments and data flow pictures were freely mixed with the VHDL description. This was very valuable later when the VHDL had to be modified. Invariably, modifications made the VHDL less readable. Chiplet model sizes varied from 200 to 5000 lines of VHDL. The relationship between lines of VHDL and actual logic cell count were inverse. Control and sequential logic required many lines of description, but synthesized into small amounts of logic. Data flow could be described compactly with vectored signals and simple operators, but synthesized into large amounts of logic.

Since we depended on text entry for the VHDL descriptions, anything that simplified the task was welcomed. We had many logic constructs where the VHDL description consisted of some declarations, attributes and concurrent signal assignments. The information was scattered throughout the model to meet the requirements of the language syntax. It was time consuming and difficult to maintain such constructs in a large model.

To ease the entry task we developed a VHDL macro preprocessor. The preprocessor keyed on VHDL "macro" statements. The macro statements were small code generators that created the necessary VHDL statements and placed them in the proper areas of the VHDL model. Attributes were entered as meaningful keywords on the macro statements. All of the information for a particular construct was contained on the macro statement. Many of the more cumbersome structural attributes were generated automatically along with some of the more tedious logic connections. For example, test generation scan rings were connected automatically by the preprocessor. As a result of this technique the designer entered only about half the actual VHDL code required to describe the design; the remainder was generated by the macro preprocessor.

The VHDL attributes were extremely important in customizing the VHDL description to the technology. The selection of I/O books was very critical to the processor design. It was difficult to write synthesis transforms that selected the proper I/O book. Directives that specified exact book types and power characteristics were a better solution. VHDL component instantiations described each I/O book. Attributes attached to the component specified characteristics that could vary from one instance to the next. Some drivers had multiple output pins, each of which was designed to drive a specific output impedance. The choice of driver pins was done with an attribute. Physical location of I/O books and powers level were also specified this way.

Logic Synthesis

The VHDL description for each chip was synthesized into logic using the Logic Transformation System (LTS). LTS is an internally developed synthesis system we have used since 1985. The VHDL was synthesized for both simulation and fabrication. However, they were synthesized much differently. For simulation purposes, a path that translated the VHDL into technology-independent logic books was used. Care was taken to correlate the logic description with the VHDL. The technology-independent books were optimized for an existing event driven simulator. Very little logic optimization was performed during synthesis. VHDL hierarchy was maintained. The primary intent of this synthesis path was to allow a designer to verify a VHDL model without the benefit of a VHDL simulator. Using the capabilities available in VHDL, this was accomplished.

The synthesis path for fabrication was much more complex. We chose to synthesize the chip logic with the internal VHDL hierarchy removed. This was a case of expediency winning out over good judgement. Our synthesis system had handled complete chip descriptions up to this point and we felt it was a lower resource cost to continue this way. Capacity projections indicated it was still possible with the target technology. We accomplished the task in this manner, but it cost us long turn-around times and loss of synthesis control. Future projects will use hierarchical synthesis as the synthesis system has been enhanced to support hierarchy. Results with this methodology are very encouraging.

Synthesis has always been a two-edged sword. On the one hand it frees a designer from the mundane and error prone task of manually converting a logic description into a hardware implementation. On the other hand it hides the implementation from the designer and makes it difficult to see where an implementation is not satisfying requirements. A good logic analysis tool that displays logic paths and timing information is a must to analyzing problems. The logic must be annotated as much as possible from the original VHDL model to help trace through it. We had several such tools that proved very valuable.

We experimented with numerous methods for influencing the synthesis tools. VHDL attributes were very useful for this. We used them at the chip boundary to define input arrival times and expected output times. The times were translated by synthesis into implementation goals. We also used attributes internally to control special logic paths. A common situation was a path that appeared to be critical to synthesis but really wasn't. Such a path was marked as a timing "dont-care". Synthesis would ignore these paths, and concentrate on real misses.

Timing misses often required a rewrite of the original VHDL description. Commonly this meant writing low level expressions for the logic. Since much care was taken in writing the description it was important to make synthesis preserve the work. We used "no-modification" and "map" attributes to control this.

A no-modification attribute attached to a VHDL signal preserved that logical point all the way to final logic. It prevented synthesis from moving logic forward or backword through the point. The map attribute was even more stringent. It forced synthesis to translate a logic expression attached to it into a technology book implementation very early in the process. This preserved the implementation through to the final logic. Both of these attributes were highly used to get final timing closure.

VHDL turned out to be a good choice for logic synthesis. It allowed the designer to rise to heights of abstraction or wallow in depths of detail. The lowest depth of detail we reached was to use VHDL functions and component instantiations; both could be written so that they mapped exactly into a technology book. We used this technique extensively for arrays and complex books that were present in the technology. It was much easier to write this type of description for complex books than make synthesis produce them some other way. If you knew what you wanted and where, this method allowed you get it.

Simulation

Simulation of the IBM AS/400 processor was successful but had to deal with several limitations. The first limitation was the simulator itself. As early users of VHDL, we had no working VHDL simulator available from either a vendor or internally. As a result, we were forced to reuse an existing gate level, event driven logic simulator. We went through three revisions on the simulator before the project ended.

The simulator we used accepted gate level descriptions as input. It also had its own behavioral language and testcase control language. We had to convert our VHDL models into a form called BDL/S (Basic Design Language for Structure). It is comparable to an EDIF net list. We developed a set of technology independent BDL/S blocks, simulation behaviors and synthesis transforms as part of the methodology. All VHDL simulation models were synthesized into BDL/S.

To make this strategy work well we had to make it easy for the designer to correlate the VHDL model to the BDL/S description. We had to extract as much information as possible from the VHDL and attach it to the BDL/S blocks and nets. We did this by making extensive use of VHDL attributes. VHDL signal names were used to generate BDL/S netnames and also signal names. The primary means of debugging a testcase is with a program called "scope". It displays waveforms for nets specified by the designer. The names chosen for nets matched the names used in the VHDL. The hierarchy present in the VHDL was maintained in the simulation models. The logical function that existed in the BDL/S also closely matched the VHDL model. With this strong correlation it was never necessary to look at the BDL/S format explicitly.

The method worked very well; we are still using it. No designer has ever looked at a technology independent BDL/S schematic. The ability to annotate the synthesized description from the VHDL allowed us to reuse an existing simulator with minimum investment. We also found that the same annotation was very useful in analyzing timing problems when synthesizing to a technology.

The design was simulated in several distinct environments. These ranged from a single chip environment to multiple chip groups up to a single system. Each environment consisted of VHDL hardware models and simulation behaviors that emulated interfaces and memory. Testcases consisted of microcode instruction streams. These had first been run on a behavioral level microcode simulation model. The results of the microcode model simulation and the hardware model simulation were compared. Discrepancies were either fixed or explained. Some self checking manual testcases were also used. The system level simulation was done with an internal hardware simulation accelerator. A model that encompassed most of the system was used along with system level microcode. A good portion of our initial power on sequence and many of the diagnostic functions that would be required in lab debug were simulated here.

Each design group had to generate chip level test patterns for chip manufacturing. They also were responsible for fixing timing problems. These tasks were done outside of simulation with standard level sensitive scan design (LSSD) test generation programs and internal static timing tools. The results were analyzed and if changes were required, they were made at the VHDL level and resynthesized. VHDL attributes were used to supply the proper test and timing flags required by the programs.

Conclusion

The chip design methodology used on the second generation IBM AS/400 processor was a mixture of new and old. It was the first processor design to successfully use logic synthesis and the VHDL design language. We were able to use the features of VHDL to mesh nicely into existing design tools. The reuse of existing tools made sense. Every new tool required extensive debug and enhancement before it was productive. Old tools usually required modifications, but they became operational much quicker. VHDL was a more difficult design language to use, but the leverage it provided was worth the extra effort.

Logic synthesis gave us improved productivity by getting the design into working logic easier. We were able to get into simulation faster and implement design changes more easily. This was especially true in the area of finite state machine logic. Synthesis could not do the job completely. It is very important to be able to guide synthesis towards a good solution. VHDL helped us do this productively. Using a combination of synthesis and designer directives we achieved an implementation as good as a manual design. Furthermore, since the design is not locked tightly into a specific technology, it becomes much easier to take the same, or "slightly" modified design, into a new technology.

Logic synthesis and the VHDL design language gave our design methodology the flexibility it needed to make the project successful. Further enhancements to the process will allow us to bring products to market faster and with fewer errors than before. This means higher quality and happier customers. That is what a design methodology is really all about.

ULTRA-RELIABLE DIGITAL AVIONICS (URDA) PROCESSOR ARCHITECTURE

Reagan Branstetter, Angela Harper, Larry Denton

Texas Instruments Incorporated
Defense Systems & Electronics Group
P.O. Box 660246, MS 3148
Dallas, Texas 75266
(214) 480–1274
FAX (214) 480–1380

ABSTRACT

The objective of the URDA program* is to develop a prototype advanced development model (ADM) processor that combines a data processor, signal processor, memory and system interface on a single SEM–E avionics card. TI's approach integrates two Ada-programmable, URDA basic processor modules (BPMs) with a JIAWG-compatible PiBus and TMBus onto a common SEM-E format ADM module and provides a separate, high-speed (25-MWord/s100-MByte/s) input/output bus for sensor data. Each BPM provides a peak throughput of 100 MIPS scalar concurrent with 400 MFLOPS vector processing in a removable multichip module (MCM). The URDA BPMs use the chipset and BPM architecture developed on the TI Aladdin program** to implement a high-performance processor in an MCM package compatible with assembly onto standard form-factor, liquid-flow-through printed wiring board (PWB) modules. A Mips R4000 family reduced instruction set computer (RISC) processor and a TI 100-MHz bipolar complementary metal-oxide semiconductor (BiCMOS) vector co-processor (VCP) application-specific integrated circuit (ASIC) provide, respectively, the 100 MIPS of scalar processor throughput and 400 MFLOPS of vector processing throughput for each BPM. Extensive software development, system modeling and simulation, and system integration and test tools are provided with the URDA processor.

URDA ADM PROCESSOR MODULE

As shown in Figure 1, the URDA ADM processor module is implemented in a SEM-E, liquid-flowthrough avionics card form-factor. The ADM module combines data processing, signal processing, memory, and system interfaces on a single avionics module. Two URDA BPMs, each with 100-MIPS and 400-MFLOPS throughput capability, are integrated with a processor interface module (PIM) to provide a single SEM-E module with 200-MIPS and 800-MFLOPS throughput, JIAWG-compatible PiBus and TMBus interfaces, and a 100-MByte/s input/output port for high-speed sensor data transfer. The URDA BPMs are repackaged implementations of the TI Aladdin BPMs. To provide such high throughput in such a small volume, they are fabricated using TI's silicon-on-silicon packaging technology coupled with commercial RISC technology and TI's advanced BiCMOS ASIC technology.[1] The URDA processor is supported by TI-developed and commercially available tools for software development, software/system integration and test, and system design.

The URDA ADM architecture is shown in Figure 2. TI's approach provides a common processor kernel implementation in an MCM package (the URDA BPMs) integrated with an interface module that provides system-specific interfaces and functionality. Each URDA BPM has an Aladdin system bus (ASB), an input/output bus (IOBUS) port for communication and data transfer, and a JTAG interface for test and maintenance control. The PIM provides a personalization of the kernel processor to the specific system and application, including the interfaces to the backplane and support circuitry for the BPMs. This approach allows the common processor kernel MCM to be used in a variety of systems and module form-factors without requiring mechanical or electrical design changes to the kernel processor functions, interfaces, or package.

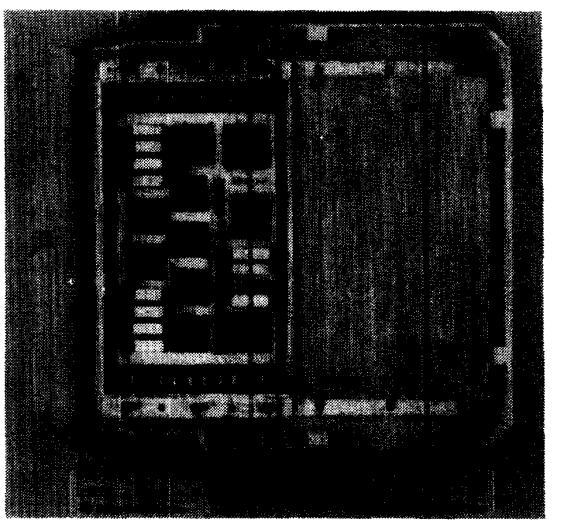


Figure 1. URDA ADM Processor Module

*Texas Instruments is developing the URDA processor under contract with the U.S. Air Force Wright Laboratory (WRDC) and the U.S. Army Night Vision and Electro-Sensors Directorate (NVESD).

**The TI Aladdin ASIC chipset was developed on the TI Aladdin program under contract with the U.S. Army Communications and Electronics Command (CECOM) and sponsored by the Advanced Research Projects Agency (ARPA) with technical direction from the U.S. Army Night Vision and Electro-Sensors Directorate (NVESD).

CH3431-4/94/0000-0274 $1.00 © 1994 IEEE

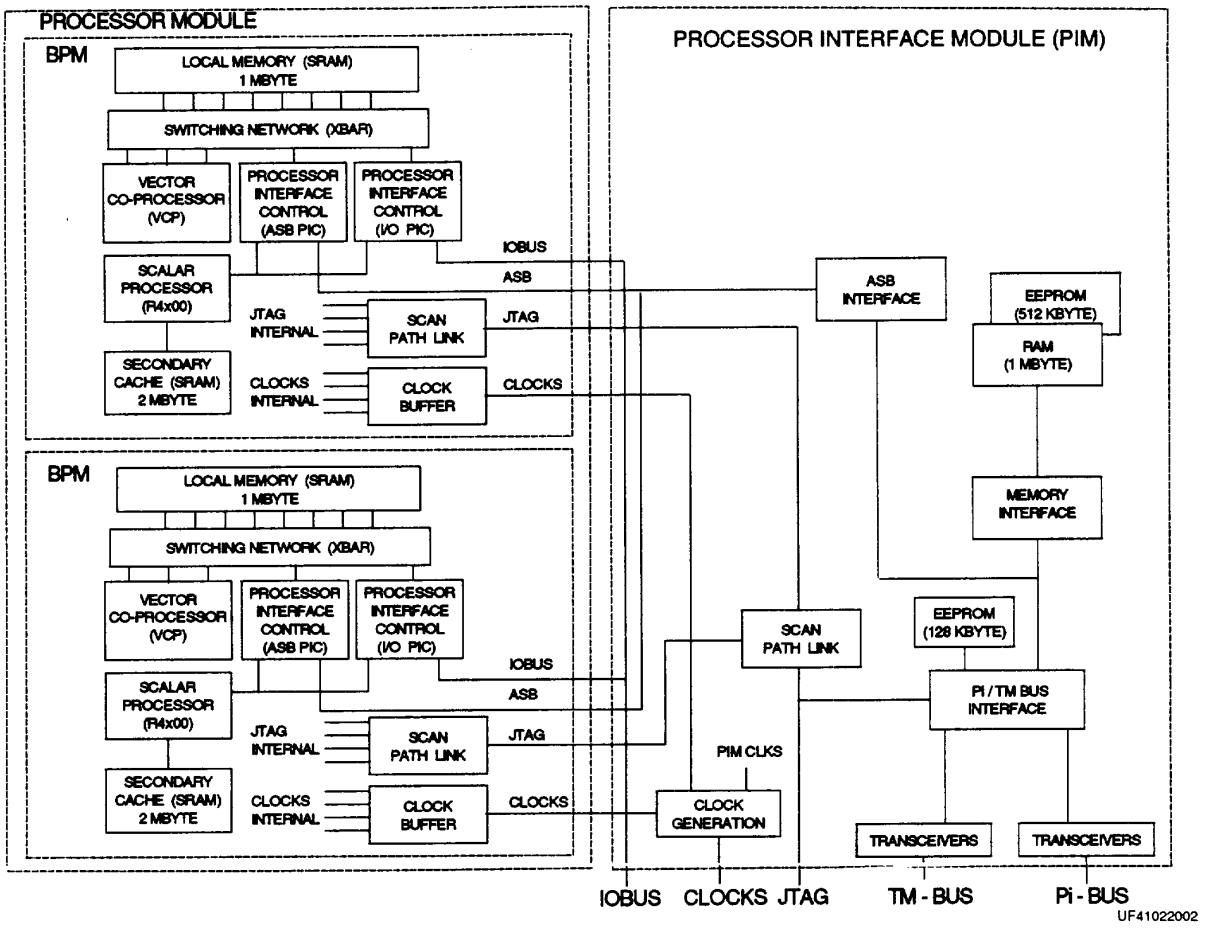


Figure 2. ADM Architecture

The URDA system requirements include JIAWG standard PiBus and TMBus backplane interfaces to the ADM and a separate 100-MByte/s, high-speed IOBUS for sensor data transfer. The IOBUSs are routed to the PIM backplane connector, providing a common 100-MByte interface to both URDA BPMs. Circuitry on the PIM implements the PiBus and TMBus interfaces to the backplane, a TMBus to JTAG translation, a separate JTAG interface to the backplane, clock generation, interrupt and discrete control support, on-module nonvolatile memory for program store, and on-module random-access memory (RAM) that acts as a store and forward interface between the PiBus and the BPMs. Both BPMs interface to the PIM store and forward memory via a common ASB interconnect. Because of design information availability and schedule compatibility, the F–16 modular mission computer (MMC) JIAWG PiBus and TMBus chipset was selected for use on the URDA PIM. A field-programmable gate array (FPGA) provides the interface between the ASB and the local memory bus of the MMC chipset.

BPM ARCHITECTURE[2]

The URDA BPM architecture is shown in Figure 2. The local memory is implemented using TI custom 16K × 9 static random-access memory (SRAM) devices designed for 100-MHz synchronous operation. Local memory is divided into eight 32-Kword (32 bits plus byte parity) banks, accessible via six crossbar processor ports. The VCP has access to the local memory via three independent data ports connected to the crossbar devices. Access to local memory from the scalar processor and the two 25-MWord/s (100-MByte/s) buses (the ASB and IOBUS) is provided by the two processor interface control (PIC) devices. The sixth crossbar processor port is configured to provide real-time monitor access via probe points to internal crossbar transactions. All activity can be monitored by external test equipment at the 100-MHz BPM clock rate, clock by clock, for any memory port or any processor port, selectable via JTAG. A commercial scan path linker device provides the interface

between the JTAG port and the JTAG test rings internal to the BPM. Each ASIC, except for the SRAM, is designed with JTAG logic for test, debug, and fault isolation to the device level on the BPM. The scalar processor, a Mips R4000, provides 100-MIPS scalar processor throughput and accesses local memory at a 100-MWord/s burst rate (in eight-word cache blocks) via either of the PIC devices. Also provided for the R4000 are 512 Kwords (2 MBytes) of secondary cache SRAM. Buffering and fanout of the 100-MHz clock is provided by a TI clock distribution circuit (CDC) that has been designed to minimize clock skew across multiple clock driver outputs. The VCP, PIC, crossbar, SRAM, and CDC devices are all semi-custom BiCMOS devices designed for the TI Aladdin program and fabricated using TI's EPIC2B 0.8-micron process.

VECTOR CO-PROCESSOR (VCP)[2]

The VCP is a microprogrammable, pipelined processor containing multiple arithmetic resources, as shown in the block diagram in Figure 3.[1,3] All resources are controlled by an on-chip microsequencer executing from a 256 instruction by 192-bit, on-chip microcode RAM.

Two 32-bit multipliers and two 32-bit arithmetic logic units (ALUs) provide a peak processing throughput of 400 MFLOPS/MOPS at a 100-MHz BPM clock rate. The arithmetic resources support 32-bit integer fixed point and single precision IEEE–754 floating point formats. A four-stage pipeline implementation allows 100-MHz operation of the arithmetic units. A 32-word register file is available for use by the arithmetic units. The data paths also support passing data between the data register file and the address register file.

Three bidirectional data ports (coupled with three independent address generators) provide a peak I/O capacity of 1200 MBytes/s. A 16-word by 24-bit address register file supports operation of the three address generators. Three index registers, a loop counter, and an accumulator register provide extensive indexing and window addressing capability. Each address generator provides a 24-bit address plus control signals to the crossbar devices, supporting sustained 100-MHz memory accesses on all three VCP ports.

Two loop counters are available for nested looping and can be loaded from microcode fields or the data register files. A program counter stack supports eight levels of nested subroutines, and several sources can be used to provide branch destinations for conditional or unconditional branching. Conditional branch decisions can be based on 30 arithmetic resource conditions, as well as loop counter status.

A command parameter list (CPL) controller provides on-chip operating system functionality in hardware. The CPL controller handles loading and execution of microcode routines as directed by Command Parameter Lists[3] implemented as application data structures loaded into the local BPM memory. Code executing on the R4000 scalar processor builds the appropriate CPLs and initiates action by the CPL controller. CPLs consist of two words and identify to the CPL controller the

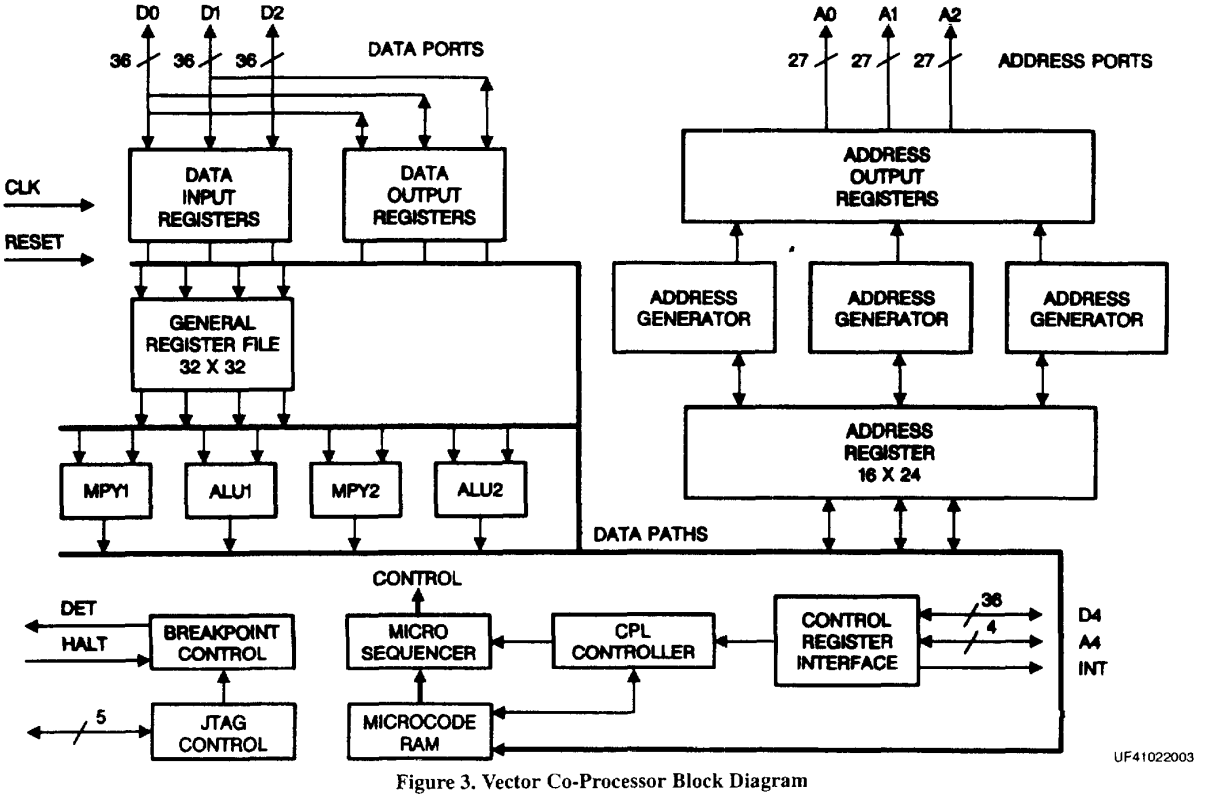


Figure 3. Vector Co-Processor Block Diagram

type of command to be executed (application, jump, microcode download, or microcode read) and can be chained together for execution of successive microprograms with no intervention required by the R4000 scalar processor.

Special consideration has been given to support for testability and software debug in the VCP design. A JTAG interface provides full boundary scan, internal scan of over 1300 data path flip-flops, and software access to debug support hardware. A signature register and signature compare logic in the VCP also provide support for software built-in test (BIT) routines and a microcode download checkpoint. Breakpoint logic consisting of two 20-bit counters with separate microcode address comparators provides multiple-event detection on microcode instruction access. The breakpoint logic is initialized via JTAG and allows event detection on "n" occurrences of address 'A' followed by "m" occurrences of address 'B'. Once the VCP is halted, JTAG commands can single-step VCP execution, access register contents, and resume operation.

PERFORMANCE[2]

With four on-chip arithmetic resources, the VCP has a maximum processing throughput capability of 400 MFLOPS executing at a 100-MHz clock frequency. This performance far exceeds the capabilities of other existing processors. Table 1 shows the performance of the VCP for some typical signal processing algorithms as compared to the performance of two popular digital signal processors (DSPs)the TI TMS320C40 and the INTEL i860. A general-purpose library of algorithms implemented in microcoded routines is available for use in developing programs for execution in the VCP.

SOFTWARE SUPPORT[2]

TI's URDA software toolset supports URDA target parallel processing applications by providing:

- Application partitioning/mapping support
- Code translation tools for the R4000
- Code translation tools for the VCP
- Run-time environment (i.e., operating system support)
- Debug support.

Application partitioning/mapping support is provided through either the ADAS toolset or Teamwork/SIM tools. A uniprocessor Ada toolset is used to generate code for the scalar processor, and a VCP toolset is used for generating microcode for the VCP. The runtime environment provides the basic operating system (OS) functions such as multitasking, interrupt handling, Ada exception handling, timer management, and virtual memory management. Additionally, the URDA runtime environment includes support for initialization, VCP interface, and interprocessor messaging. Programs for either the R4000 or the VCP can be downloaded independently to the target processors for execution and debug or can be downloaded, executed, and debugged as cooperating programs in a multiprocessor system.

VCP PROGRAMMING[2]

The VCP tool suite (Figure 4) is centered around the TI Register Transfer Language (RTL) toolset, which provides the core assembler, linker, simulator, and bit generation capabilities. Source RTL programs for the VCP can be written directly or, with the aid of a VCP-timing-file-to-RTL (VTR) two-dimensional programming utility or, with the aid of an Ada-to-microcode compilation system. VCP object files can be reverse assembled into two-dimensional source code using an RTL-object-to-VTR (RTV) or can be converted to Ada packages for interfacing to R4000 Ada programs using RTL-object-to-Ada (RTA). Each VCP programming tool is described in more detail below.

1. Ada.—The JRS Research Laboratories Inc. (JRS) Ada compilation system is VAX-hosted and allows writing VCP programs in Ada, testing the Ada implementation using the VAX Ada debugger, and compiling the Ada to produce VCP microcode in the form of a source RTL program. The compiler supports all VCP hardware features, including multiple arithmetic units and multiple address generators. Built-in functions have been implemented to allow the Ada programmer to use specialized VCP hardware features such as window addressing, bit reverse addressing, and special ALU and multiplier opcodes. As a result, the Ada programmer can write very stylized code, which will typically result in more efficient microcode. Average execution time ratios between compiler generated microcode

TABLE 1. VCP PERFORMANCE COMPARISON

Algorithm	VCP (ms)	TMS320C40 (ms)	Intel i860 (ms)
Complex fast Fourier transform (FFT), radix 2, 1024 points	0.23	1.55	0.74
Vector inverse, 1024 points	0.08	0.36	0.06
Vector multiply and add, 1024 points	0.01	0.05	0.12
Convolution, 128 × 128 array, 5 × 5	2.5	25.4*	13.1
3 × 3	0.7	13.5*	4.6
Matrix multiple 128 × 128	21.8	110.6*	77.2
Vector square, 1024 points	0.01*	0.05*	0.06
Histogram, 1024 points, 8-bit data	0.03*	0.35*	0.50

*Estimated timings

- AUTOMATES INCLUSION OF MICROCODE RELATED INFORMATION REQUIRED BY SCALAR PROCESSOR ADA PROGRAM
- PROVIDES PROCEDURE CALL INTERFACE FROM SCALAR PROCESSOR ADA TO MICROCODE

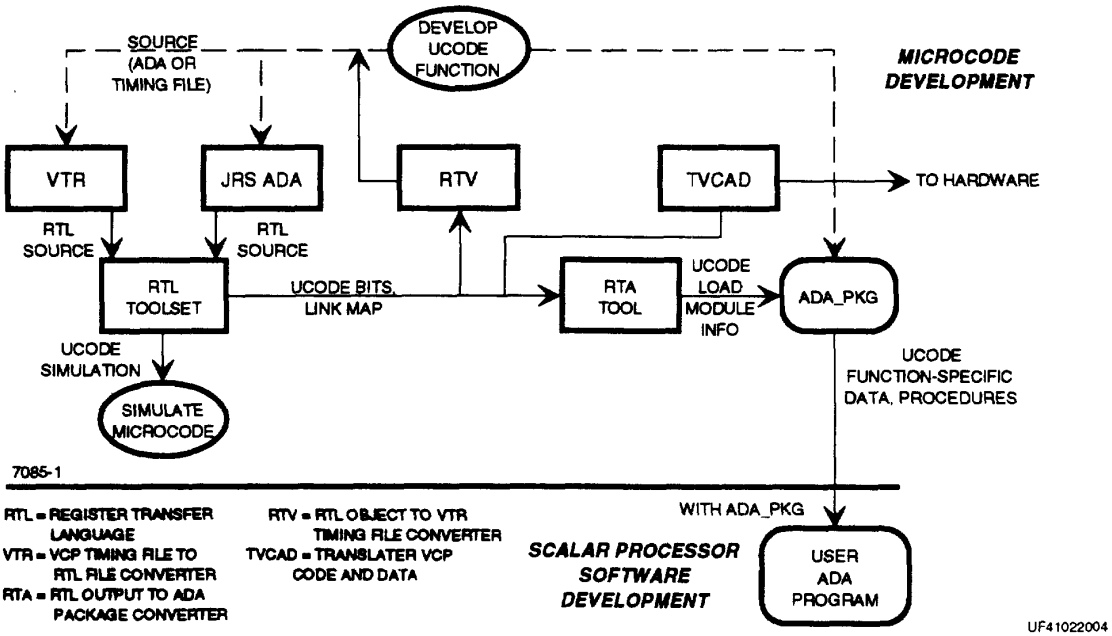


Figure 4. Aladdin Ada Interface to Microcode

and hand-generated microcode have proven to be 2.2 for stylized Ada and 5.0 for unstylized Ada.

2. RTL.—The RTL tools are also VAX-hosted and allow for complete development- and instruction-level simulation of the VCP with its associated memory. The RTL tools let system architects and designers define a custom programming language, simulation environment, and bit generation utilities for any programmable part, not just the VCP.

3. VTR.—The VCP-timing-file-to-RTL file converter is a VAX-hosted tool that simplifies the generation of handwritten VCP microcode by providing the VCP microprogrammer with a bird's-eye view of the two-dimensional code being developed. It also allows VCP code to be written using shorter mnemonics than those supported by the RTL toolset. These two things combined will result in more efficient hand-generated VCP microcode.

4. RTV.—The RTL-object-to-VTR source translation utility executes on the IBM PC in interactive or command line mode and disassembles VCP object files to produce two-dimensional VCP source code files. Information about the number of loops, length of each loop, and efficiency of each VCP part within each loop is written to the bottom of each file. This utility was designed to analyze VCP microcode generated by the JRS Ada compiler. It also has been used to provide a two-dimensional view of hand-generated microcode that was not generated using VTR.

5. RTA.—The RTL-object-to-Ada package translation utility is VAX-hosted and reads VCP object files, VCP link map files, and user preference information to generate an Ada package for interfacing VCP applications to R4000 Ada. RTA uses the link map file, microcode slice file, and a modified simulation control file output by the VCP RTL toolset. RTA reads the modified simulation control file and link map file to generate an Ada record type representing the CPL chain. The VCP link map provides CPL names and link addresses while the simulation control file provides CPL execution order, which dictates CPL chain record structure. Load module information is extracted from the VCP link map and translated into Ada load module constants including start address and length of each load module and of each CPL within each load module.

6. TVCAD.—The translate VCP code and data utility converts VCP object files, CPL chain, and data information into one file that can be downloaded into the hardware. The download file contains three columns of information including address, data to be written at the specified address, and a comment pertaining to the type of data being written.

Ada PROGRAMMING[2]

Programming for the URDA ADM processor is based on a uniprocessor programming paradigm with either shared memory accesses or message passing providing the inter-BPM communications. Applications targeted to the URDA architecture are partitioned statically into the software units allocated to

each logical BPM. Each partition allocated to the scalar processor on a BPM is developed using a traditional uniprocessor Ada toolset, as shown in Figure 5. Using the RTA tool described previously, R4000 code used to initialize data structures for VCP routines can be generated automatically for inclusion in the scalar processor program. Additionally, any data packages that are global to the BPMs within an URDA processor may be "with'd" into the application program for each of the BPMs. Then, standard compilations/assemblies generate object modules that can be linked with the necessary runtime and OS primitives to form a load module.

Likewise, the VCP toolset generates microcode for execution on the VCP. The microcode can be included as data within the scalar processor program load module or may be loaded independently under user control to the appropriate BPM local memory. From the Ada program on the scalar processor, the VCP can be invoked using either blocking or nonblocking remote procedure calls. Communications between applications on different BPMs are supported through shared memory access or through message passing primitives provided by the runtime environment.

RUNTIME ENVIRONMENT[2]

The TI URDA ADM runtime environment is based on a TI-produced Ada runtime and executive developed on the TI Aladdin program to support embedded, real-time applications targeted to the R4000 scalar processor. In addition to a standard Ada runtime, the TI URDA runtime supports uniprocessor counting semaphores, dynamic modification of task priorities, additional timer support (delay until a future time, read and set system time), uniprocessor locks, and physical/virtual memory management routines. The physical/virtual memory management routines allow a virtual address/window to be assigned dynamically to a physical address and also allow for the physical address associated with a virtual address to be returned. Support also is included in the runtime environment for BIT, application software interfaces to the VCP, and inter-BPM messaging. The BIT capability includes processor startup, system initiated, and periodic BIT functions, which are used for fault detection and isolation.

The VCP interface function manages the invocation of the microcode from the Ada program executing on the scalar processor. To the scalar processor Ada programmer, the interface appears as either blocking or nonblocking remote procedure calls. This allows for either polling or event-driven synchronization between the R4000 and the VCP. The interface also provides access to the VCP status and configuration registers.

The messaging function provides task-to-task communications, where tasks send messages to "message queues" that may be assigned or shared by different tasks. The messaging function is an R4000 target/ASB bus implementation of the TI Common Network Operating System. Extensions to the Aladdin messaging function comprehend inter-ADM communications across the PiBus using the store and forward memory of the PIM. Messages may be sent logically, without requiring the sender to know who or where the destination is. Application tasks may

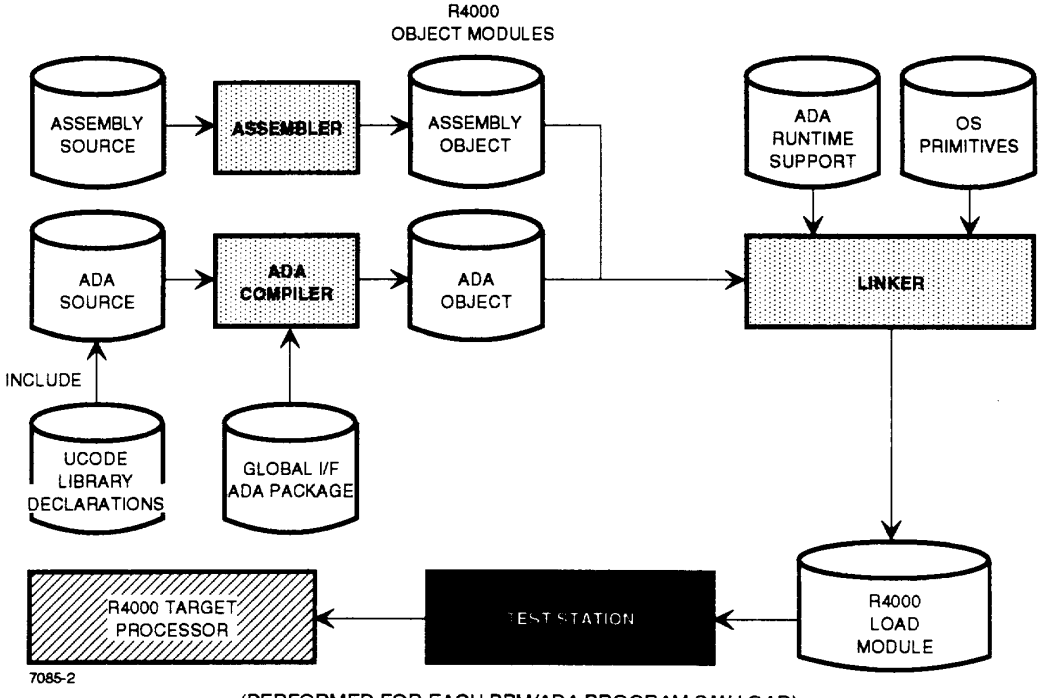


Figure 5. Aladdin R4000 Code Development Flow

request messages from a queue in either a blocked or nonblocked mode. The blocked receive mode allows the application task to be blocked until either a message is received or a specified time-out occurs. The nonblocked receive mode allows the requesting task to poll for receipt of a message. The store and forward memory on the PIM allows PiBus messages to be received into the PIM memory. Interrupts notify the scalar processors that a message has been received into the store and forward memory and is available for processing. Messages targeted for transfer across the PiBus are sent to the PIM memory first; then a PiBus transfer is initiated by the scalar processor to send the message from PIM memory to the appropriate PiBus receiving node. Both read and write PiBus transfers can be initiated, allowing data to be sent from or fetched into the initiating ADM.

INTEGRATION AND TEST SUPPORT[2]

Extensive integration and test support is provided for the URDA processor. Test and debug tools provide debug support at several levels. Support is provided for system-level functions, BPM debug, VCP debug, and scalar processor debug support.

System-level functions include utilities for downloading VCP and R4000 programs, as well as supporting software and hardware test and debug. The BPM debugger provides BPM-level control and supports VCP test and integration. The BPM debugger provides access to local memory and allows user access to the ASB and the IOBUS from the BPM under debug control. Breakpoint management also is provided at the BPM level. VCP instruction execution, crossbar processor or memory port transactions, and ASB or IOBUS activity can be defined as events to be detected. VCP operation then can be halted upon detection of breakpoint conditions. The VCP debugger allows the user to halt VCP execution for stepping through CPL execution at the instruction level or the CPL level (i.e., clock by clock or on CPL boundaries). While in a VCP halt state, access is provided for inspection and modification of internal VCP register values. The SP debugger is based on the Mips R3000 debug monitor. Debug monitor code executes on the target, providing program execution control, access to memory and variables, and breakpoint management.

SUMMARY

TI's URDA processor is ideally suited to advanced radar and signal processing applications. The TI URDA processor combines data processor, signal processor, memory, and system interface functions on a single SEM-E module. Advanced BiCMOS ICs, silicon-on-silicon packaging, and commercial RISC scalar processor technologies have provided the capability to implement a single SEM-E avionics card form-factor signal processor with 200-MIPS/800-MFLOPS throughput. The TI URDA processor's modularity, scalability, Ada programmability, high performance, and extensive software development and debug support environment provide a powerful, general-purpose processor for use in Department of Defense (DoD) embedded signal processing systems.

REFERENCES

1. Branstetter, Reagan; Chuck Roark; and William Ruszczyk. "Aladdin Processor and Software Support," *1990 Digest of Papers* for Government Microcircuit Applications Conference (GOMAC), November 1990.

2. Branstetter, Reagan; Angela Harper; and Larry Denton. "Aladdin Signal Processor and Architecture Modeling," *SPIE Conference ProceedingsArchitecture, Hardware, and Forward-Looking Infrared Issues in Automatic Target Recognition,* Volume 1957, April 1993.

3. Hearn, Alan; Dirk Luckett; Michael Perry; and William Warner. "A 400-MFLOPS Vector Co-Processor and Aladdin Operating System Interface," *1992 Digest of Papers* for Government Microcircuit Applications Conference (GOMAC), November 1992.

TA 6.1: A 100MHz Macropipelined CISC CMOS Microprocessor

Roy Badeau, R. Iris Bahar, Debra Bernstein, Larry Biro, William Bowhill, John Brown, Michael Case, Ruben Castelino, Elizabeth Cooper, Maureen Delaney, David Deverell, John Edmondson, John Ellis, Timothy Fischer, Thomas Fox, Mary Gowan, Paul Gronowski, William Herrick, Anil Jain, Jeanne Meyer, Daniel Miner, Hamid Partovi, Victor Peng, Ronald Preston, Chandrasekhara Somanathan, Rebecca Stamm, Stephen Thierauf, Michael Uhler, Nicholas Wade, William Wheeler

Digital Equipment Corporation, Hudson, MA

This macropipelined CISC microprocessor is implemented in a 0.75µm CMOS 3.3V 3-metal-layer technology. The 1.3M-transistor custom chip measures 1.62x1.46cm^2, dissipates 18W (peak), and is packaged in a 339 pin PGA. The chip implements a macroinstruction pipeline to execute the instruction set of a popular CISC minicomputer. Figure 1 depicts a block diagram of the major functional units. Figure 2 shows a die micrograph.

The IBOX fetches and decodes macroinstructions, and contains the instruction parser/operand decoder, a 2kB direct-mapped virtual-instruction cache, and a dynamic branch prediction unit. The microsequencer and EBOX work together as a 4-stage micropipeline to execute instructions under microprogrammed control. The control store is a 1600 61b microword ROM. The microcode can be patched using a CAM to substitute RAM words for ROM. The FBOX is a 4-stage pipelined floating-point and integer multiply execution unit. Most floating-point instructions have a latency of 4 and a repetition rate of 1 cycle. The MBOX processes memory references. The 2-way set-associative 8kB write-through primary cache (PCACHE) is in the MBOX. Looking up a physical address from the 96-entry fully-associative TB, accessing the PCACHE, and rotating and passing the PCACHE data to the requester takes 1.5 cycles. PCACHE operations take 1 cycle. The CBOX controls the off-chip 2nd-level direct-mapped write-back cache and maintains multiprocessor cache coherency using ownership protocols. The BIU is the interface to the external bus connecting the chip to the memory subsystem.

The added parallelism of the macropipeline combined with the larger caches and more efficient compilers yields a 2.4-fold improvement in TPI over that achieved for a fully micropipelined microprocessor with the same CISC instruction set [1].

The chip has 2 sets of synchronous clocks: 100MHz 4-phase internal clocks, and 4-phase external clocks that run 3 times slower and control the BIU, pad logic, and peripheral chips. The chip is functionally insensitive to overlap between adjacent phases. For performance, it is critical to keep skews small and edge rates sharp, so special attention is paid to clock generation and distribution (Figure 3). A 400MHz ECL oscillator drives the receiver buffer at the top of the chip. The output is routed to the global clock generator (CLKGEN), a 400MHz 12-state FSM, at the center. The internal (divide by 4) and external (divide by 12) phases are derived by decoding these states. The outputs of the phase decoders are synchronized by flip-flops to eliminate generation skew (Figure 4). The outputs of the flip-flops are buffered by 4 inverters and distributed, using the 3rd metal layer (M3, 16mΩ/sq.), in the central clock routing channel that extends the height of the chip. Clocks are supplied to the boxes by tapping off the central clock routing and buffering each signal with 4 inverters. This buffering minimizes loads seen by the clocks in the routing channel, where RC delays are held to 30ps. The buffered clocks are then distributed east and west, again using M3 but strapped with M2. Before the clocks are used, they are inverted locally.

These final stages of buffering reduce the loading on the east-west clocks and sharpen the clock edges. Conditional clocks are generated without a delay penalty using 2-input gates for the local buffers. The layout for the clock buffers and routing is balanced: the buffer sizes are tuned to the extracted parasitics and dummy loads are added to the more lightly-loaded phases. An on-chip decoupling capacitor (30nF) is distributed throughout the buffer layout to minimize Vdd/Vss noise when the clocks switch rapidly.

The clock signals were measured on silicon by E-beam probe (Figure 5). The clock skews and edge rates across this 1.62cm chip are less than 0.5ns and 0.65ns, respectively.

Dynamic logic and differential cascode circuit techniques were used to achieve the 100MHz clock rate. For example, cascode logic is used in the FBOX double-precision hardware divider. The divider is a radix-2 SRT divider with overlapped partial remainder calculation (using carry save adders) and quotient bit selection (QBS). Figure 6 shows the most important QBS circuit in the divider critical path. The circuit is a cascode implementation of a 3b carry chain plus sum logic (the XOR stage) for one bit. Logically, the circuit outputs the sign bit (T&C) of the estimated partial remainder. The propagation delay, from the least significant G0 (generate) input to the sign outputs, is simulated at 1.2ns for typical parts operating under worst-case conditions.

The design was extensively analyzed for logical correctness and electrical integrity before fabrication. 1 billion cycles were run on the RTL model. 75 million cycles were run on the gate-level model (abstracted from transistor netlists). The operating system was also booted on this model. 600k cycles were run on the switch-level model. A 3-D capacitance extractor was used to determine the load on every node. A trapezoid-based extractor was used to determine node resistances: clock nodes had over 100k elements. The extracted Rs and Cs were fed to a static timing verifier which traversed 350k signal paths and checked 42k timing constraints on a 500k transistor netlist in a single run. CAD tools were used to identify potential design and reliability problems: coupling, latchup, charge-sharing, noise, race conditions, hot carrier stresses, and EM.

The chip booted the operating system at 100MHz on 1st silicon. The patchable control store proved invaluable: the few obscure bugs that slipped through verification were easily fixed by patching the microcode. 100MHz parts are benchmarked at 50SPECMARKS, out-performing all previously reported CISC microprocessors.

Acknowledgement

The authors acknowledge contributions of W. Anderson, E. Arnold, B. Benschneider, M. Benoit, D. Bhavsar, M. Blaskovich, P. Boucher, L. Briggs, S. Butler, R. Calcagni, S. Carroll, A. Cave, S. Chopra, M. Coiley, A. DiPace, D. Donchin, D. DuVarney, R. Dutta, H. Fair, G. Franceschi, N. Geagan, S. Goel, J. Grodstein, W. Grundmann, H. Harkness, R. Hicks, C. Holub, R. Khanna, R. Kiser, D. Koslow, K. Ladd, S. Levitin, S. Martin, T. McDermott, K. McFadden, B. McGee, M. Minardi, T. O'Brien, J. Pan, N. Phillips, J. Pickholtz, R. Razdan, S. Samudrala, J. Siegel, K. Siegel, J. StLaurent, R. Supnik, M. Tareila, S. Watkins, Y. Yen

Reference

[1] R. Allmon, et al., "CMOS Implementation of a 32b Computer," ISSCC DIGEST OF TECHNICAL PAPERS; pp. 80-81; Feb., 1989.

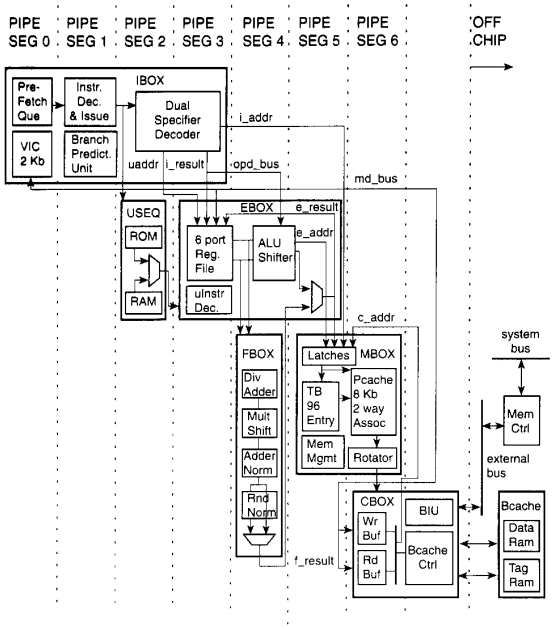


Figure 1: Block diagram showing macropipeline segments.

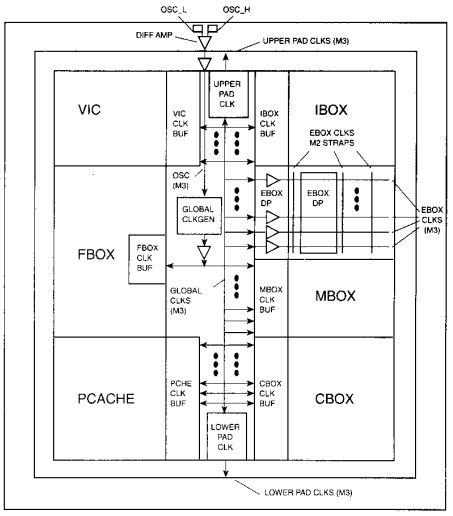


Figure 3: Clock generation and distribution scheme.

Figure 2: See page 255.

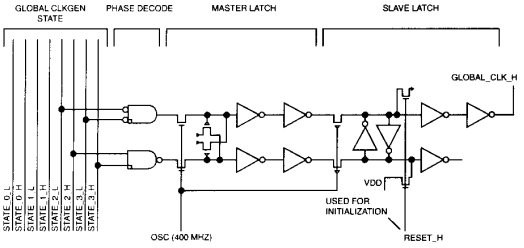


Figure 4: Clock phase decode and synchronizing flip-flop.

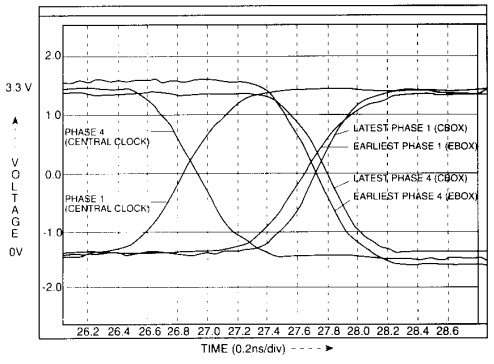


Figure 5: On-chip clock waveforms from e-beam probe.

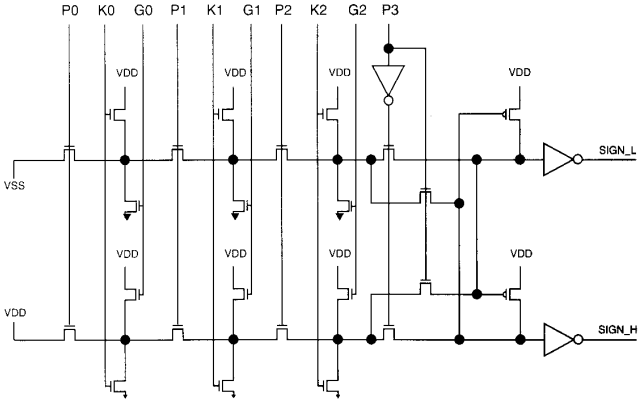


Figure 6: Estimated partial-remainder sign-detection logic in FBOX divider.

High Performance Computer Systems

ES/3090: A REALIZATION OF ESA/370 SYSTEM ARCHITECTURE IN IBM's MOST POWERFUL MAINFRAME COMPUTER THROUGH A BALANCE OF TECHNOLOGY AND SYSTEM INNOVATIONS

WILLIAM J. NOHILLY
MANAGER "PROCESSOR SYSTEMS DESIGN"

IBM CORPORATION
KINGSTON, NEW YORK

Abstract

This paper describes how IBM's most powerful system achieved significant growth through a balance of technology exploitation and systems design. Selective use of a new advanced transistor technology combined with innovative processor design techniques solved many system design limitations.

This paper will describe several system design problems and the solutions. The paper will describe how a broad spectrum of ESA environments were enhanced and how significant data path improvements resulted in very high throughput and fast response time.

The ES/3090™ is the newest and most powerful S/370 Processor. It is a system that was designed using the current 3090 series as a base with the objective of implementing the new Enterprise Systems Architecture. The key objective of the architecture is to relieve virtual addressing constraints and provide system managed address spaces to relieve user management of storage.

To exploit ESA/370™ the ES/3090 System has been specifically redesigned to provide a balance of performance vs bandwidth. In the initial stages of the design, the simulation runs showed that increasing the IER (Internal Execution Rate) of each processor by cycle time reductions and CPI (Cycles Per Instructions) improvements resulted in less and less of this performance being delivered to the customer at the system level. This was more noticeable at the 4-way and 6-way processor system configurations.

Key areas had to be significantly redesigned to provide improvements in a broad spectrum of ESA environments.

I will describe how these system design areas in the ES/3090 were redesigned by using a balance of new technologies and new system data paths and packaging.

Writable Control Store:

The ES/3090 is a horizontal microcode controlled processor with multiple data fields to allow simultaneous usage of the independent functional elements (Parallel Adder, Serial Adder, Shifter, High Performance Multiply and General Purpose Register Operations). (Figure 1 shows the relationship).

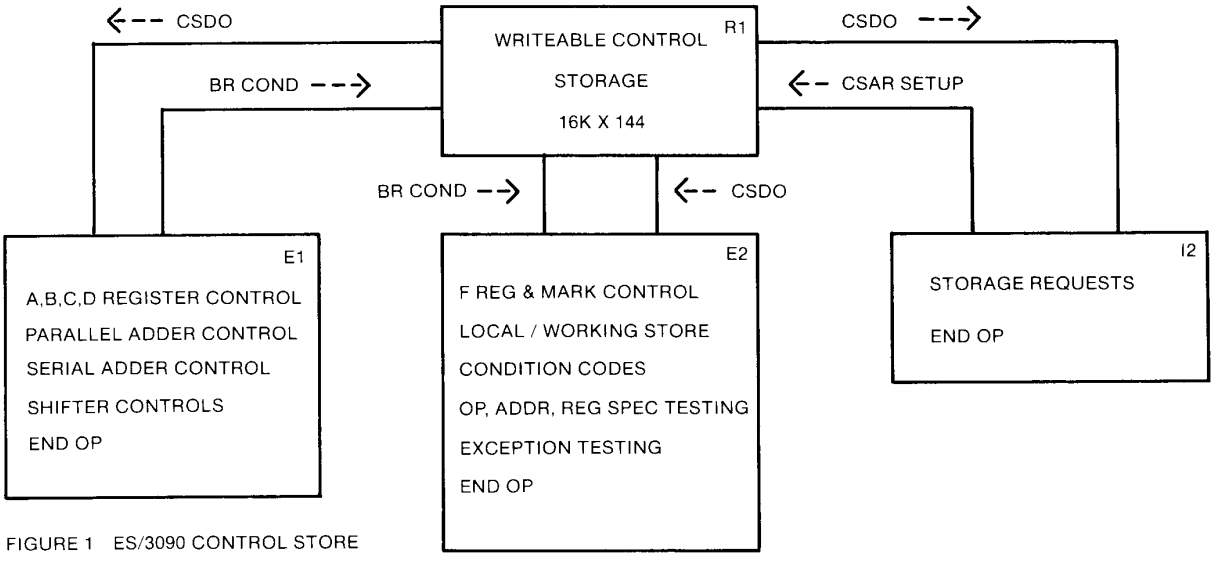


FIGURE 1 ES/3090 CONTROL STORE

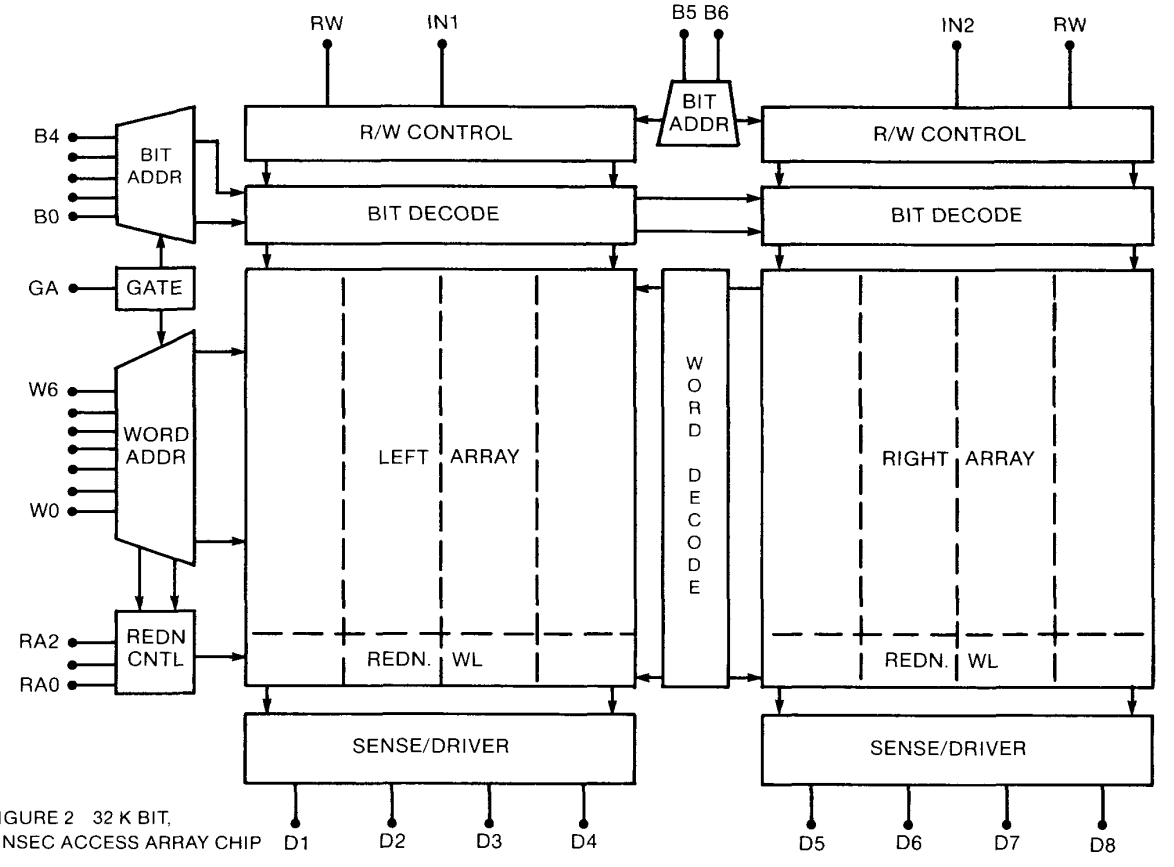


FIGURE 2 32 K BIT, 3 NSEC ACCESS ARRAY CHIP

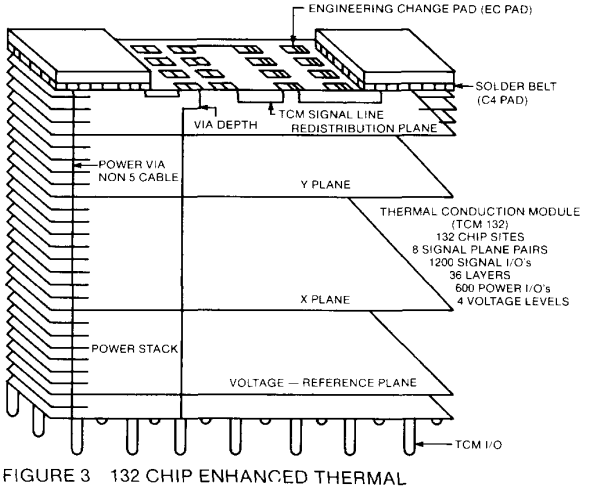


FIGURE 3 132 CHIP ENHANCED THERMAL CONDUCTION MODULE

The ESA/370 required a large writeable control storage to allow the hardware design to proceed in parallel with the architecture development. Key to this implementation is a new advanced 32k bit chip capable of achieving a one cycle access inside the 15 nanosecond cycle time. This chip was specifically designed and personalized to fit into the Thermal Conduction Module. Figure 2 and 3 are slides showing the 32k bit, 3 nsec access chip and the enhanced 132 site Thermal Conduction Module.

A key feature of the writeable control store is that this allows ES/3090 to participate in future architecture enhancements without requiring hardware changes in the field.

Active Address Spaces

The ESA/370 architecture introduces the concept of access registers to relieve virtual addressing constraints. Implementation of ESA/370 in the ES/3090 is via hardware registers managed by a set of 6 new architected instructions that provide users 15 - 2 gigabyte virtual data spaces. In addition, segment table origin address registers and new address translation implementation schemes were required. Figure 4 shows how this resulting address space expansion is made available to the user.

To efficiently manage this increased capability, ESA/370 introduced the Linkage Stack Architecture to manage the change of state of the machine. This required significant hardware and microcode design enhancements in the Execution Elements, the Instruction Elements and a new Writable Control Store.

ESA/370 implementation is an extremely efficient and very secure architecture and provides a smooth transition from XA architecture by implicit usage of the register using

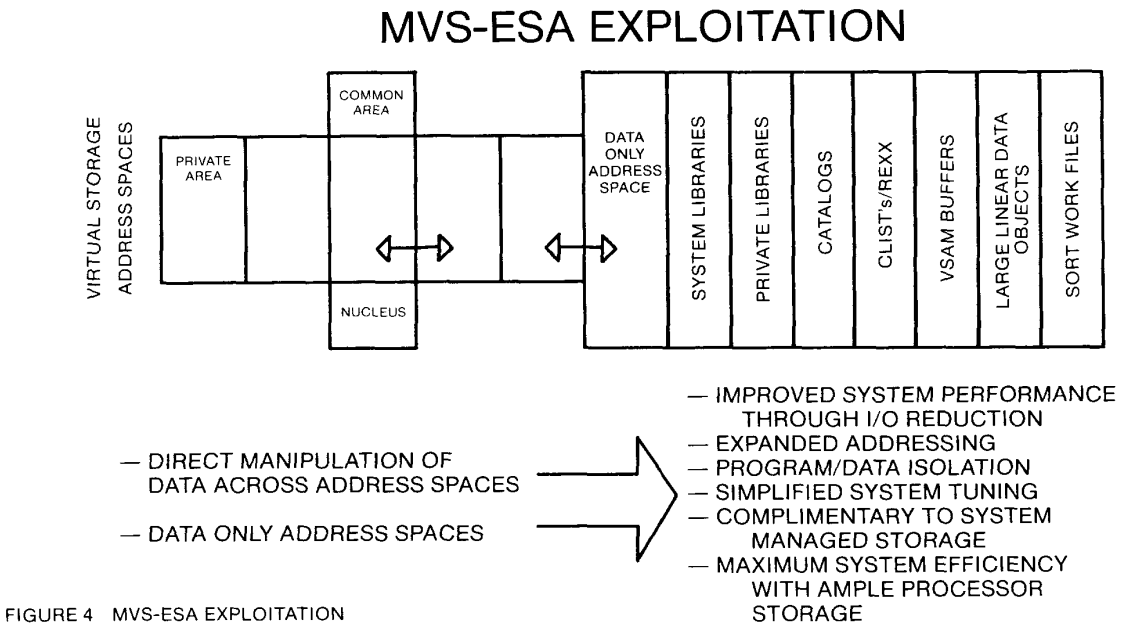


FIGURE 4 MVS-ESA EXPLOITATION

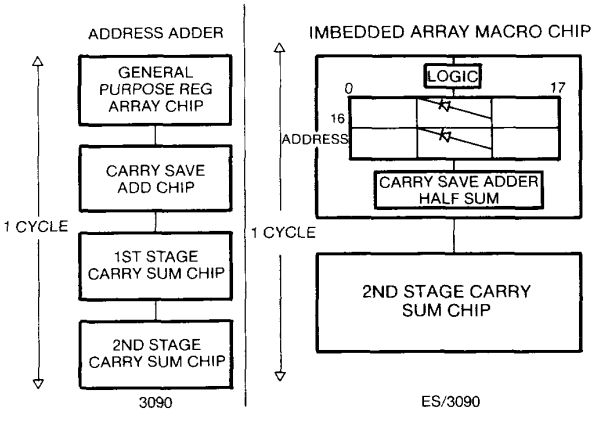


FIGURE 5 IMBEDDED ARRAY LOGIC CHIP

the Data Spaces. This also allows programmers to move across data spaces without resetting the state of the machine.

A key to the implementation of active address space is the capability to have "buried array structures" inside the new LSI logic chip. Figure 5 shows the implementation of these type registers in prior systems and in the ES/3090. The key advantages are in reduction of chip crossings, integration of logic function into the array chip and increased reliability by reducing pins. Without this functional advantage it would not have been possible to achieve the levels of density necessary to package all of the new architecture inside the Thermal Conduction Module and fit in the 3090 packaging structure. This packaging results in the 3090S Models being upgradeable from the 3090 and 3090E thus allowing the customer to continue to maximize the investment in the 3090 product line.

Data Sharing Performance Improved

To provide the maximum benefit to ESA environments, the Data Paths and Capabilities of the 3090 were greatly expanded. New simulation techniques provided insight into the system areas that required additional bandwidth with the new ESA environments. For example, early in the design cycle, a performance simulation model that operated cycle by cycle showed that when the CPI (cycles per instruction) was reduced via improved functional units, the request rate to memory increased so significantly that the MIP's (Internal Execution Rate) of the 4-way and 6-way were then actually lowered. As the performance model continued running a steady state throughput rate resulted which was significantly lower than our expectations. Armed with this knowledge from the beginning, the design engineers set about to design a balanced system to achieve the increased data sharing performance requirements of ESA/370. Figure 6 shows the ES/3090 Model 600S which is the most powerful commercial processor in existence today. Major enhancements included:

— Increasing the size of the store-in high speed buffer to 128 kbytes to effectively increase the instructions per MISS of all 6 caches.

— An improved cache management scheme that enhances the utilization of the cache to cache transfer mechanism.

— Increasing the size of Main Storage to 512 mbytes and expanded storage to 2 gigabytes to service six processors and reduce paging rates.

— Improved storage access by utilizing IBM's proprietary 1 megabit chip with an access time of 80 nsec.

— Increased memory bandwidth

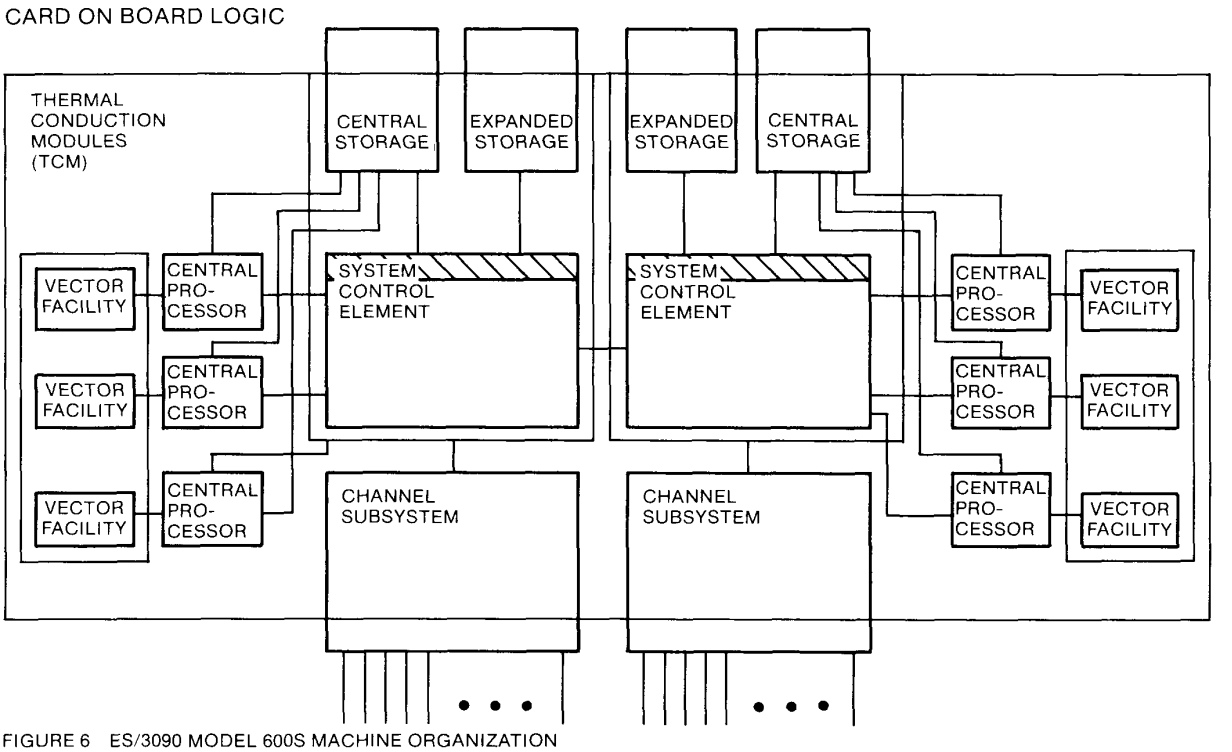


FIGURE 6 ES/3090 MODEL 600S MACHINE ORGANIZATION

Additional paths to the memories provided a 2x improvement in overall bandwidth; this allowed overall IER (Internal Execution Rate) improvements to result in very high throughput and fast response time. In addition, several new buffering techniques were invented to allow the System Control Element which contains the three independent cross-point switches to be dynamically assigned for less time to each of the memory elements. The data transfer times between System Control Elements in a Multi Processor configuration were improved significantly.

System throughput is achieved not only by processor cycle time and processor functional enhancements but also by a systems total configuration and the efficiency with which software and hardware work together. ESA/370 together with the balanced system design of the ES/3090 provide improved throughput at up to 1.5 times for the Model 600S versus the Model 600E in a MVS/IMS environment.

Summary and Conclusion

I will be glad to answer your questions on the ES/3090 system but I would first take a few minutes to describe the machine organization, the chip structures, the thermal conduction modules and the system upgrade paths.

The ES/3090 achieves its design objectives not only by improved VLSI technologies but by careful attention to designing a balanced system structure at all packaging levels. The resultant system substantially exceeded the initial design goals and by achieving up to a 50% growth in performance at the high end, it is clearly the most powerful commercial processor.

Acknowledgements

The author would like to extend our appreciation to the many individuals who provided creative ideas, suggestions and innovative system and technology implementations. Special appreciation is for the dedicated Processor System Development design team in the IBM Kingston Laboratory and for the semiconductor designers in the IBM East Fishkill Laboratory whose technical expertise provided very innovative chip design solutions.

The author would also like to acknowledge the Multiprocessor design enhancements of Messrs. R. D. Herzl, P. M. Moore, and R. N. Fuqua of the IBM Kingston Laboratory and the prior cooperative work with B. U. Messina of the IBM Poughkeepsie Laboratory.

References

1. S. G. Tucker, "The IBM 3090 System; An Overview," IBM Systems Journal, Vol. 25, No. 1, 1986.

2. B. W. Curran, L. J. Milich, R. D. Herzl "Memory Controller (P2) Buffer", U. S. Patent Pending.

3. Y. H. Chan, J. L. Brown, R. H. Nighuis, C. R. Rivadeneira, J. R. Struk, "3 nsec 32K Bipolar RAM", presented at the 1986 IEEE Solid State Conference.

4. L. Liu "An Improved Multiprocessor Cache Using Locally Changed State," U.S. Patent Pending.

IBM's ES/9000 Model 982's Fault-Tolerant Design for Consolidation

Consolidated work loads running around the clock means that today's large, general-purpose computers must meet high availability demands. To meet these demands, the Model 982 provides fault tolerance by combining enhanced circuit-level error detection and failure isolation techniques with system-level techniques exploiting inherent redundancy.

Lisa Spainhower

Thomas A. Gregg

IBM Large Scale Computing Division

Ram Chillarege

IBM T.J. Watson Research Center

Recent trends in high-end general-purpose computing point to a need to consolidate commercial information-processing applications and databases. Although this need appears to run contrary to the continuing growth of distributed computing, the commercial world has compelling reasons for consolidating data centers and ultimately computers. These reasons include acquisitions and takeovers, elimination of data centers, and staff reductions. Consolidation reduces costs and simplifies and improves system management and data security. At the same time, however, it increases demands on the remaining data center computing equipment. Thus, the data center computer needs to have very high performance and capacity and fault tolerance that provides close to 100 percent availability.

Computing has become a multiple-time-zone operation or even a global operation. With consolidated computing, an organization may have work loads running in prime shift at some part of the globe at any time. A related trend is the elimination of off-prime or nonprime shift operations—applications such as credit card authorization, automated tellers, and point-of-sale terminals must be on line all the time. As a result, a computing system can never take a scheduled outage. Consolidation, coupled with the nature of today's applications, places stringent availability requirements on high-end computers.

Another result of consolidation is the simultaneous existence of unrelated work loads and multiple operating systems on the same computer. When applications are added because of acquisitions or site reductions, one larger computer can replace several of lesser capacity without a need to change the applications. The logical partition for each operating system is customized for the capacity of the computer it replaces. Since one computer supports all the partitions, each of which could have a different critical operations window, the overall demand for hardware availability is greater, spanning all the windows. To satisfy this demand, the fault tolerance design objective for IBM's ES/9000 Model 982 computer is to keep running without outage or intervention either at the time of failure or during repair.

Design philosophy

Our general-purpose computer architecture has evolved since 1964 from the System/360, through the System/370 and the System/370 Extended Architecture (XA), to today's System/390 Enterprise Systems Architecture. The current hardware implementation is the ES/9000, which includes three machine types: the 9221, 9121, and 9021, from least to highest performance. The high-end 9021 is available in models with from one to eight CPUs. The most powerful is the eight-CPU Model 982.

Fault tolerance is implemented at two distinct levels in the 982. At the circuit level, a set of objectives and guidelines for consistent error checking and recovery ensures fault tolerance

0272-1732/94/$04.00 © 1994 IEEE

throughout the computer, regardless of each subsystem's function. At the system level, on the other hand, subsystem functions determine fault tolerance implementation. From a wide range of fault tolerance techniques, the designer selects the best for each subsystem function. Both the circuit and system levels must meet overall product requirements, not just for fault tolerance but also for cost and performance.

Circuit-level issues. A computer's error detection and recovery strategy must handle the most frequent circuit fault models. Logic errors can occur due to hard, intermittent, and transient circuit faults. The hard, or permanent, fault occurs when a digital circuit no longer yields a correct output, given a specific set of inputs. Any time the specific input is repeated, the circuit produces the incorrect output. An intermittent fault occurs when a specific event produces an incorrect result, but the same inputs at a different time may produce the correct result. An intermittent fault can occur as a result of a design error or marginal circuits. Transients occur when environmental conditions, noise, or cosmic particles cause an incorrect result, but the circuit itself functions correctly.

The expectation for the chip technologies used in the 982 is that most faults are transient, so the design focus is to retry operations and recover at the hardware level whenever possible. Once the design has been debugged and the computer installed in a stable environment, intermittent failures are extremely rare. The computer is not specifically designed to handle intermittents. For infrequent intermittents, instructions are retried and recovered; if such errors occur frequently, they exceed thresholds and are considered permanent faults.

The IBM S/360, S/370, and S/390 computers have been known for their data integrity. Traditionally, when an error occurred, these computers checked all computation, retried the operation, and if it was unrecoverable, the computer stopped and perhaps a higher level of fault tolerance (such as software) handled the situation. Consequently, very extensive concurrent error detection was designed into the computers.

To make the 982 fault tolerant, its designers increased the traditional concurrent error detection from the 90 percent range to 100 percent. An error is usually identified and recovery attempted within the same machine cycle. Reporting and recording may take additional cycles. A code protects every latch, and there are no naked latches. All dataflows, arrays, and control buses include parity or ECC, and state machines use techniques such as parity predict or duplication. Expanding to 100 percent coverage added less than 3 percent to the number of logic chips. This increase is so small mainly because many of the hard-to-check logic chips, such as sequence logic, are very limited in pins, and thus the logic can be completely checked without adding chips.

In addition to concurrent error detection of all logic, on-line error correction and repair capability requires exact identification of the replaceable part that causes an error. For recovery, the computer also should identify the source of the erroneous data. As an example, the design ground rules require that error checkers be placed at the driver of any signals leaving a part and immediately after the receiver of any signals entering a part. This allows most failures to be identified to the correct part without further analysis. On-line failure isolation determines the part to be replaced.[1] The system must be able to determine whether an error is due to a permanent physical failure requiring repair or only a transient fault that can be handled without physical part replacement. In a 982, it is not always easy to distinguish which type of fault has occurred, but the design handles all of them.

Distinguishing faults requires the capability to back out and retry internal operations with appropriate thresholds for determining success. Since retrying an operation can involve different paths through the logic than the original error, the system of thresholds is very fine grained and rather complex. For example, if a fault occurs on a directory entry in a cache, the retry of the operation may use a different cache set. Therefore, the threshold must be on the use of the directory address and set, not on the actual operation being performed. This has led to a very extensive threshold implementation.

Failure due to a hard circuit fault might be recoverable through instruction retry since the machine is run nonoverlapped during the retry, thereby using different circuit combinations. Intermittent faults, on the other hand, could cause errors several times and go away later. To deal with either type of fault, multiple retries with appropriate thresholds determine whether a unit needs to be taken out for repair or whether the computer can continue with the unit containing the failure. Chen et al. give a detailed description of the 982's error detection techniques.[2]

System-level issues. The 982 has three major functional subsystems: the CPUs, the channel subsystem, and the storage hierarchy. It also has a fourth subsystem consisting of the support functions: the processor controller element (PCE), power, and cooling. Together, these four subsystems are called the central processing complex (CPC).

For performance, the CPC has several identical elements in parallel, such as processors and channels. These inherently redundant resources provide a fundamental fault tolerance capability. Each element is designed at the circuit level to have concurrent fault detection, recovery by retry, and fault identification and isolation. The strategy for fault tolerance is to couple the error detection and failure isolation capabilities with system-level techniques exploiting the inherent redundancy in the CPC. As a result, the 982 can identify errors and recover transient and intermittent failures by retry.

The storage hierarchy passes back detected errors to the CPU or channel. Returning these "passed errors" to the

continued on p. 52

Overview of the ES/9000 Model 982

Figure A shows a functional layout of the Model 982. The collection of hardware subsystems—CPUs, storage hierarchy, channel subsystem, and support subsystem, including all licensed internal code—that make up a 982 is known as the central processing complex (CPC). The 982 is a tightly coupled multiprocessor. It has eight CPUs and can be physically configured as one eight-way or as two four-way multiprocessors. Each four-way "side" is electrically independent with no common failure points between the two. In either physical configuration, each multiprocessor can be logically divided into as many as 10 operating system partitions. Each logical partition has either shared or dedicated CPUs and I/O channels and dedicated central and expanded storage. Its relative size and priority are defined to accommodate the operating system's work load and response needs. These characteristics can be dynamically changed, resulting in very efficient use of the multiprocessor's capacity.

Each CPU has a 256-Kbyte cache divided into a 128-Kbyte data cache and a 128-Kbyte instruction cache. The CPU also has three standard execution elements, one for hardware-performed general-purpose instructions, one for hardware-performed floating-point instructions, and one for microcoded instructions. Additional thermal conduction modules (TCMs) can be installed for optional cryptographic or vector execution elements. The elements can execute instructions concurrently with one another.

Each of the two system control elements (SCEs) has a switch for data transfer on the buses between the level 2 cache and the eight CPUs, all of which can have storage operations simultaneously in process. The SCE also includes four quadword (128 data bits each) level-2-to-level-3 buses. The SCEs contain the level 2 caches, which hold copies of all the store-through level 1 data caches. Each 4-Mbyte level 2 cache has an affinity with its local 1-Gbyte central storage and is a subset of the data contained therein, although level 2 may be a more recent copy since it is a store-in cache to central storage. Data created for central storage is maintained in the CPU in an ECC-protected buffer until it can be stored in level 2, which is also ECC-protected.

The SCE provides control functions for level 2 and central storage; some of these are addressing, cache coherency, and data protection mechanisms. The two SCEs are cross-configured to all the processors as well as to the switches to the channel subsystems and expanded storage. The data throughout the storage hierarchy has fault-tolerance and correction capability.

Each side of the 982 has an interconnect communication element (ICE) containing a

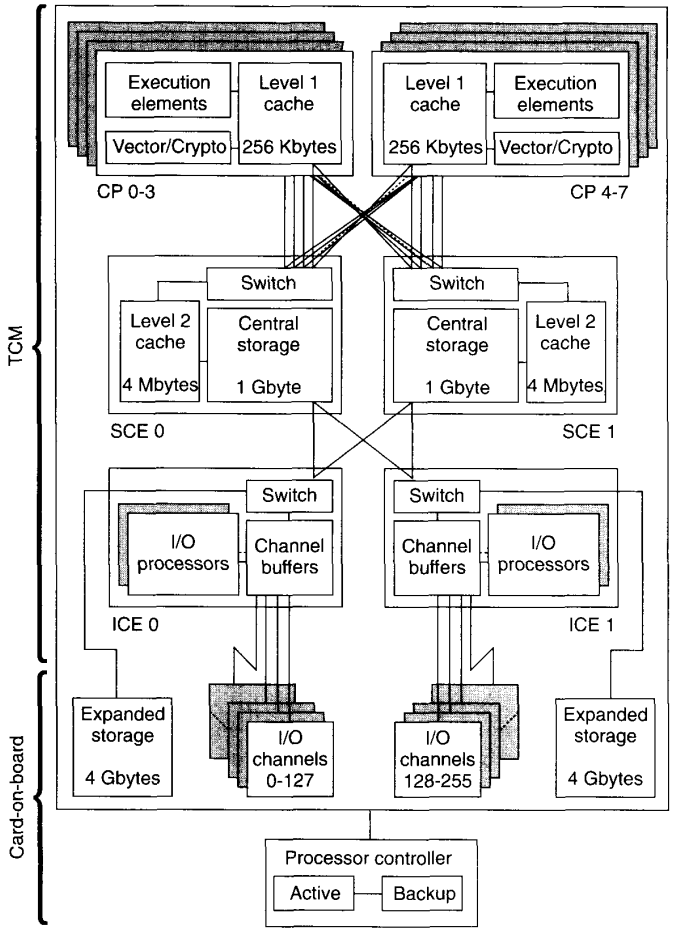


Figure A. Functional layout of Model 982.

Overview of the ES/9000 Model 982 (continued)

switch that can communicate with either SCE. The switch also attaches to a controller for expanded storage and to the I/O channel subsystem via the channel buffers. The channel buffers regulate storage requests from the channels and provide a connection to the I/O processors, more formally called the integrated offload processors. The IOPs serve as backup for one another. The channel buffers also attach the individual channels to a communication path to the IOPs. The IOPs are a RISC design known as the 801.[1]

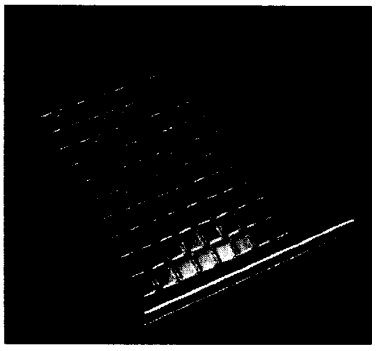


Figure B. Thermal conduction module.

Figure C. Model 982 ESCON channel card.

The 982 contains up to 256 individual channels with either parallel electrical I/O interfaces or serial optical ESCON interfaces. The parallel interfaces operate at up to 4.5 Mbytes per second at a distance of up to 400 feet. The ESCON interfaces operate at 200 Mbits, allowing data transfer at up to 18 Mbytes per second, at a distance of up to 30 kilometers using the extended distance feature.

The CPUs, SCE, and ICE all use thermal conduction module technology. A TCM is a 127.5-mm-square glass-ceramic and copper substrate that carries up to 121 ECL (emitter-coupled logic) chips. The 63 metalized glass-ceramic layers interconnect the chips and supply power. The whole package is mounted in a metal jacket that facilitates cooling. The TCM is water-cooled. The cooling mechanism consists of a piston and a heat sink in contact with the chip. The package is filled with helium gas to improve conductivity. Figure B shows a TCM substrate with chips attached. TCMs are mounted on a 60×70-cm, 22-layer, glass-epoxy board for their interconnection and power. The SCE board contains six TCMs, and the ICE and CPU boards contain four TCMs each.

The remainder of the CPC is air-cooled and packaged in card-on-board technology. Central storage is packaged on a TCM board and expanded storage on a printed circuit board. In both cases, the major component on the cards is a 4-Mbit DRAM chip.

Both the parallel and ESCON channel cards are also packaged on printed circuit boards. Figure C shows a Model 982 ESCON channel card with an optical duplex jumper cable attached. At the lower right is an uncapped ESCON channel logic module, exposing the three logic and three array chips. Each ESCON card contains two physically separate channels. The ESCON card without the optical connectors is about 120 mm wide and 180 mm long. The two rectangular boxes next to each optical connector are the transmitter and receiver, which are connected to the serializer/deserializer modules.

The logic modules, measuring 50 mm on a side, are the largest components. Their 21-layer ceramic substrates are almost 4 mm thick. Each logic chip is fabricated by means of the IBM CMOS2 process.[2] The chips contain random logic and a number of embedded arrays. The array chips are CMOS static arrays containing 144 Kbits each, with an access time of 15 ns. A recent improvement in the ESCON channel replaced the logic module with a single chip (12.7 mm square) fabricated with the IBM CMOS4 process. This chip reduces cost while increasing the reliability of the ESCON channel.

The 982 is packaged in 21 frames, ranging in footprint from 0.73 square meters to 1.38 square meters. Twelve of the frames house the functional subsystems, and the remaining nine house the support subsystem. Frame height varies from 1.8 to 2.0 meters, and the entire CPC weighs 15,560 kilograms.

References

1. J. Cocke and V. Markstein, "The Evolution of RISC Technology at IBM," *IBM J. Research and Development*, Vol. 34, 1990, pp. 4-11.
2. A. Aldridge et al., "A 40K Equivalent Gate CMOS Standard Cell Chip," *Proc. IEEE Custom Integrated Circuits Conf.*, IEEE Computer Society Press, Los Alamitos, Calif., 1987, pp. 248-251.

continued from p. 49

requester limits the scope of transient faults to a single operation and allows unaffected storage operations to continue. The computer can tolerate many permanent faults, particularly those in array chips, by reconfiguring the array to prevent use of the failed location. For permanent faults requiring repair, the CPC isolates the failing element, reconfigures the system around it, and recovers by rolling back to an error-free consistent state, all without stopping the work load. The design provides the capability to fence off all external interfaces to an element so that reconfiguration and maintenance can be performed while the computer continues in operation. A service technician can replace the failed hardware without affecting the rest of the CPC.

It is crucial that the PCE and the rest of the support do not impinge on the availability of the functional subsystems. Because the support subsystem does not have the performance constraints of the functional subsystems, it can use more traditional fault tolerance techniques such as explicit redundancy—that is, duplication strictly for reliability. The storage hierarchy also employs well-documented methods for data protection; the multiplicity of methods and the resulting design robustness exceed most implementations. For these reasons, we describe the support subsystem and the storage hierarchy only briefly in the following sections. We describe the CPUs and channel subsystem in more detail because they best illustrate the system-level design direction.

The central processor

All S/390 CPCs with more than one CPU can dynamically vary any CPU between on-line and off-line status, and, as long as one CPU is on line, continue instruction execution. Except for special operations (vector facility and integrated cryptographic facility) for which hardware may not be provided on all CPUs, each CPU in a multiprocessing environment can execute any task dispatched by the operating system. Where a particular task will be executed is unpredictable.

The 982 multiprocessing environment is usually one eight-way multiprocessor but may be two four-way multiprocessors, also known as a physically partitioned multiprocessor. Any S/390 multiprocessor can have one native operating system or up to 10 logical partitions. Each logical partition has an independent operating system with some number of logical CPUs assigned to it. The number of CPUs per partition cannot exceed those physically available. Logical CPUs can be dedicated or shared. Any dedicated CPUs can be assigned to only one partition, thus reducing the maximum available to coexisting partitions. For redundancy, a partition is shared or has at least two dedicated CPUs.

A major design goal is to exploit this capability, allowing the computer to continue uninterrupted operation if a CPU becomes unavailable. Accomplishing this goal requires the following procedure:

1. The processing state in the failed CPU must roll back to a consistent, error-free state.
2. The state of the storage system as seen by all other CPUs must be architecturally accurate and free of errors.
3. The PCE must ascertain the failed CPU's state (transient failure or permanent failure).
4. The PCE must remove the failed CPU from the configuration before it propagates any corrupted data.
5. The operating system must transfer the work that was running on the failed CPU to an operational processor.
6. The failed CPU must be repaired and returned to service without a system interruption.

An example involving instruction retry and recovery illustrates the implementation of this procedure. Figure 1 shows the pipeline stages and four issued instructions. This machine implements out-of-sequence execution and exploits this performance technique to boost fault tolerance at the circuit level. Out-of-sequence execution allows multiple instructions to be issued without requiring that they finish in the order of issuance as long as they complete in the same order. There is a fine distinction between *finish* and *complete*: *finish* means reaching the end of the instruction's execution; *complete* means storing the instruction's results. Essentially, once an instruction completes, it cannot be backed out because its results are posted in storage and architecturally the instruction has executed. Prior to completing, an instruction can be backed out, provided all pending stores from instructions issued after it are also canceled and the storage is left in a consistent state.

Assume a failure occurs at time T2 as shown in Figure 1. The clocks are immediately frozen at time T2. Note that the last completed instruction was instruction 1, which completed at time T1. Although instructions 3 and 4 have both finished (their results have been calculated and put into temporary facilities awaiting completion), they cannot be flagged as completed since instruction 2 has not completed. Since the machine maintains instruction execution results in temporary facilities until it determines the instruction is complete, it can back out of all intermediate pipeline operations (including the finished instructions) by flushing the pipeline and temporary facilities. This leaves the executing program in an error-free processing state—that is, at a point corresponding to the completion of instruction 1. Assuming a failure in the CPU hardware, the physically separate PCE examines the pipeline contents, architectural facilities, and temporary facilities, and puts them back into a consistent state. The PCE obtains the CPU state by scanning out the internal facilities, using totally separate scan clocks so that the logic clocks can remain stopped.

Next, the PCE must ensure that the storage system is consistent. The storage hierarchy design facilitates this step. An advantage of its store-through level 1 data cache is that it

isolates internal CPU faults. Because the shared level 2 cache contains any critical system data resulting from CPU instruction execution, the level 1 cache copy is not critical and can be discarded upon detection of an error.

Instruction retry determines whether the failure detected was a transient error or a true permanent circuit failure. The CPU register manager,[3] which controls out-of-sequence instruction execution, provides an audit trail of changes effected by incomplete instructions; the audit trail is used for checkpoint restart in instruction retry. To perform retry, the PCE sets the CPU's internal logic state to the state it would be in at the completion of instruction 1. The PCE then steps the logic clocks to see if the following instructions execute successfully with no errors. If they do, a transient error exists, and instruction execution recommences. During retry and for some time after, instruction execution is nonoverlapped. Each instruction must complete before the next sequential instruction is decoded. If retry is successful, normal overlapped instruction execution will start.

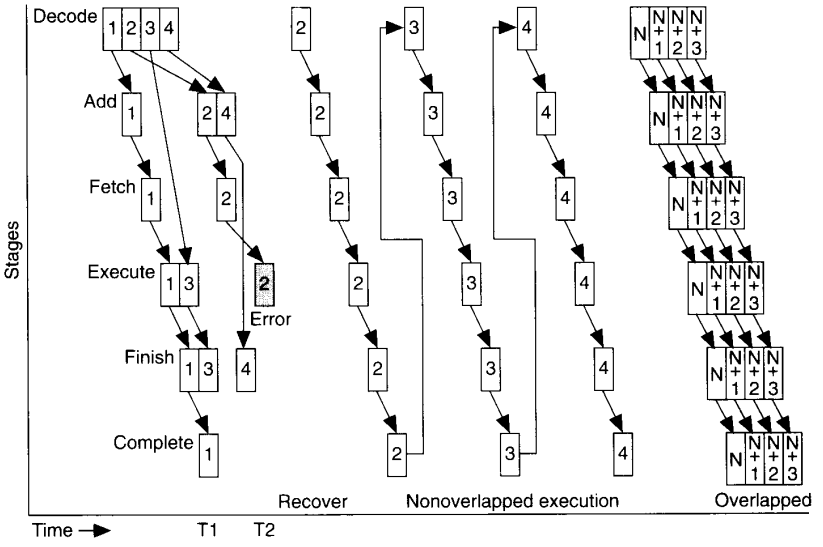


Figure 1. Out-of-sequence instruction recovery (N = instruction at which overlapped execution resumes).

If retry is unsuccessful after exceeding a retry threshold, the failure is permanent. The computer can tolerate many permanent faults, especially those in large arrays. Examples are the level 1 caches, their directories, a cache of previously translated virtual addresses called the directory look-aside table, and a cache of previously taken branch addresses called the branch history table. In these arrays, individual addressable locations and/or groups of locations can be deconfigured when a permanent fault is identified. For most other permanent CPU failures, the work load must be moved. When one of these occurs, the PCE puts the failed processor into the CPU check-stopped state.

CPU check-stop[4] indicates severe CPU damage and terminates processing on that CPU. In a uniprocessor, CPU check-stop is the equivalent of system check-stop, and all instruction processing ceases. In older models, when there was more than one CPU, the operating system would initiate alternate CPU recovery, which logically removes the failing CPU from the system.[5] The 982 uses the processor availability facility (PAF), which physically isolates the failing CPU for repair while the work in progress is completely recovered and run on the remaining CPUs.

Processor availability facility. PAF presents the check-stopped condition, the error status details, and the complete status of the process in execution to the operating system. Upon examination of the error status, the operating system stores the task in the normal dispatch queue to allow resumption of execution on another processor according to normal dispatch priorities.[6]

In the past, when retry was not successful, alternate CPU recovery would abnormally end (ABEND) the task and initiate software recovery. Depending on the robustness of the recovery and the criticality of the ABENDed task, the work load would or would not continue. For general control program tasks where recovery is quite robust, recovery was successful. If the task was an application, it probably would not recover, but its termination generally would not stop the work load. However, for many subsystem tasks and other critical component tasks, it was common for the work load to stop.

With the PAF operation, the task is not ABENDed, and no other software recovery mechanisms need be invoked. PAF provides on-line reconfiguration, which, combined with concurrent error detection and fault identification, makes the 982 a fault-tolerant processor.[7] During and after the PAF operation, $N-1$ CPUs continue to execute instructions normally, no tasks are lost, and no intervention is required. The normal dispatching priority ensures optimum use of the sys-

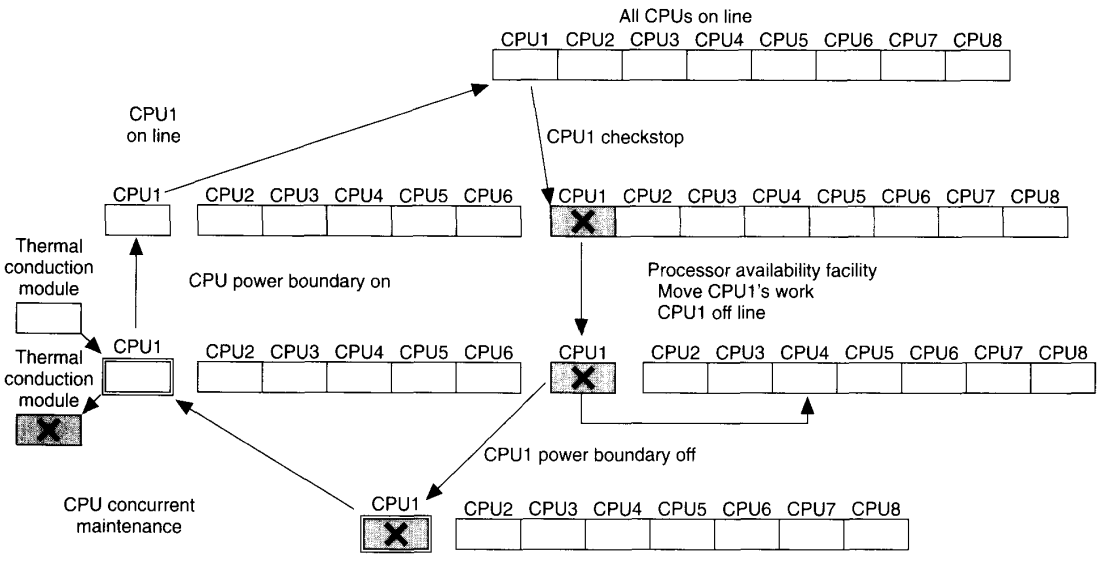


Figure 2. Concurrent CPU repair.

tem while it is running with one less CPU.

Nondisruptive repair. One of our objectives was to provide concurrent maintenance for redundant subsystems, whether inherent or explicit. Figure 2 shows the procedure for concurrent repair of a CPU. The failed CPU must be both logically and physically isolated from the operational components of the CPC. Physical isolation is provided by a power boundary. Each CPU has its own independent power supplies, which are not shared with any other hardware and which can be powered up and down independently of the power status of other power boundaries. Logical isolation is provided by fencing, an internal mechanism by which any physically connected, on-line logical unit disables the receipt of signals from a CPU. The hardware automatically invokes fencing.

The PCE also automatically invokes an auto-call for repair by a customer service engineer. At a time convenient to the customer, the engineer performs on-line maintenance: powering down the CPU boundary, replacing the failed part, powering the boundary back on, and putting the CPU back into the configuration. Again, throughout this repair process, $N-1$ CPUs continue uninterrupted instruction processing, and upon its completion, all CPUs are on line, executing tasks.

The channel subsystem

This subsystem connects the 982 to the I/O control units, which in turn connect to the I/O devices: disk and tape drives, terminals (usually personal computers), and printers. Because of the large number of channels, each of which can communicate with many devices concurrently, the channel subsystem allows recovery actions affecting as few I/O operations as possible.

When a program or operating system communicates with a device, it first builds a channel program listing I/O commands (read, write, locate, and so on), central storage addresses for data, and several flags controlling execution of the channel program.[4] Next, the CPU executes an instruction that starts the channel program by placing it on the I/O work queue. At this point, execution of the I/O operation passes to the channel subsystem and the CPU is no longer involved; it is free to do other work.

One of the I/O processors (IOPs) in the channel subsystem examines the work queue and finds an I/O operation to perform. This processor selects a path and passes the operation to the selected channel. The channel then fetches and executes the channel program.

Some operations to I/O devices take a considerable amount of time without requiring data. For example, a seek operation to a disk may take several milliseconds. To better utilize the channel path during these long operations, certain bits in the channel program instruct the channel to allow the I/O control unit to disconnect from the channel interface, allowing other operations to be initiated to other I/O control units and devices. As a result, I/O operations are multiplexed within a channel. When the I/O device finishes the opera-

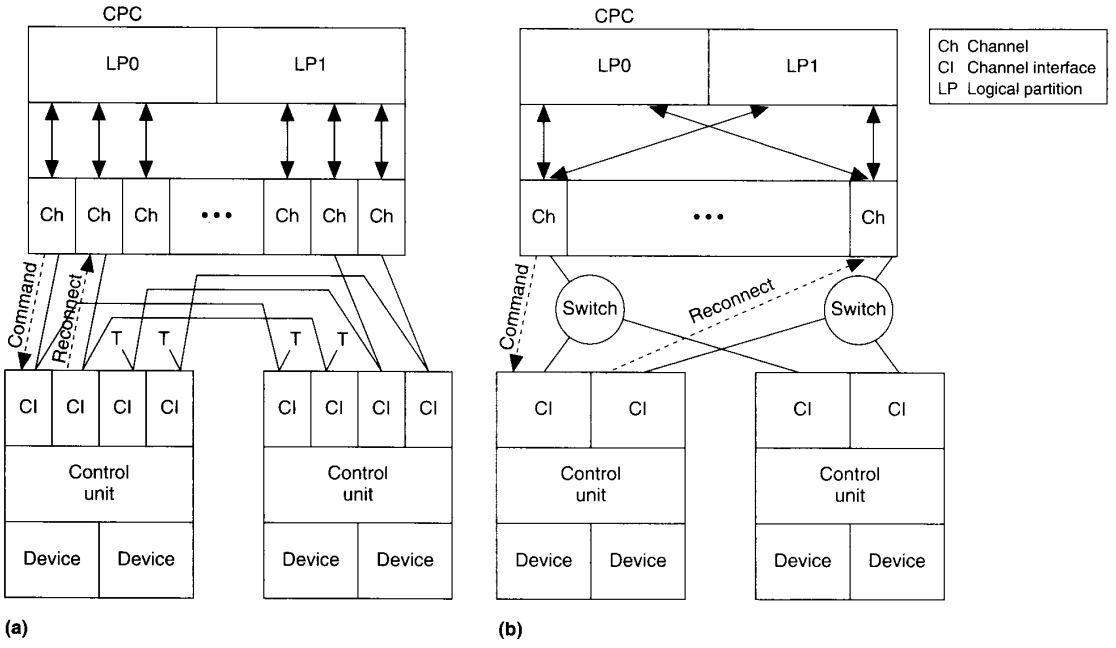


Figure 3. I/O configuration: parallel I/O interface (a); serial ESCON interface (b).

tion, it interrupts the channel subsystem over one of the available channel interfaces. This multiplexing over a channel interface requires that the channels themselves execute many channel programs concurrently. If the channel detects an error when many channel programs are active, the recovery algorithm usually affects only the single channel program actively communicating over the I/O interface.

Figure 3 illustrates how S/390s are attached to I/O control units. Figure 3a shows two control units attached to a system by parallel I/O cables; Figure 3b shows the same connection using the serial Enterprise Systems connection (ESCON) cables through two switches. Each information route from an operating system to a control unit and ultimately to an I/O device is called a path. In both examples, the CPC includes two operating systems running on two logical partitions.

The parallel I/O interface attaches to the I/O control units using a multidrop bus configuration.[8] Each of the 35 conductors (two 9-bit unidirectional data buses and 16 control lines) connects the channel to a number of I/O control units. The conductors actually pass through the I/O control units until they reach the last I/O control unit, where there is a terminating resistor (indicated by a "T" in the figure). The serial ESCON interface attaches to the I/O control units by means of pairs of point-to-point optical cables, one cable in each direction.[9,10] The switches, called ESCON directors, provide improved connectivity to the I/O control units.[11]

Dynamic path selection. Multiple paths to the I/O control units improve system performance. Before the 370/XA architecture was developed, the program or operating system was responsible for selecting the path to the I/O device. The 370/XA architecture further improved performance by moving the path selection function into the channel subsystem. Now, when an operation encounters a busy condition, the channel subsystem can choose another route to the device. The channel subsystem controls the status of available physical connections, the choice of physical connection, and the routing of operations.

Figure 3 shows that an I/O command initiated along one physical path (indicated by "Command" in the figure) can complete along a different path ("Reconnect") chosen by the I/O control unit—a capability known as dynamic reconnection. The program or operating system is aware only of the I/O devices; the channel subsystem and the I/O control unit recognize the physical connections between the system and the I/O device.

Multiple paths to the I/O control units also provide redundancy and have long been a requirement of high-end general-purpose processors. Because redundancy is inherent,

the channel subsystem can exploit it transparently to the operating system. Just as the channel subsystem can get around busy conditions, it can also get around permanent failures in a single channel or channel path, rerouting if a channel is no longer available.

The channel subsystem records status information about work queues and data in central storage. A channel that has a permanent failure between active I/O operations can be removed from the configuration without affecting any other in-process I/O operations. Dynamic reconnection, the ability of an I/O device to continue an I/O operation with an attached channel other than the one that initiated the operation, also provides fault tolerance of permanent failures in a single channel. If a channel experiences a permanent failure while actively communicating with an I/O device, the channel subsystem may continue the operation on another channel or it may request recovery of the operation from the operating system.

Channel subsystem recovery. In the simplest fault-tolerant design, errors that occur in the channel subsystem would be recovered by the program or operating system. The channel would simply indicate the error to the program or operating system, which would then carry out the appropriate recovery action. Sometimes this recovery action would be to repeat the operation. Other recovery actions might be more complicated, involving a sense operation to the I/O device to determine the state before retrying the operation.

Unfortunately, some older I/O devices and the programs that drive them cannot recover from transient errors. An example is a punched-card reader, which cannot back up and reread a card. When the reader presents an error to the program, the application is terminated, and human intervention is required to get the application going again. A similar situation exists when errors cause a loss of knowledge of the tape position in older tape drives. To handle such less-than-perfect recovery by the program, a robust recovery scheme in the channel subsystem keeps many of the errors from reaching the program. Newer I/O control units and devices, especially disk drives, are much more recoverable by software.

Since not all devices are 100 percent recoverable by software, the channel subsystem should recover from as many transient errors as is practical. Therefore, in the 982, a channel can recover from transient errors within itself and from errors on the I/O interface. The IOPs and channel buffers can also recover from transient errors; however, the recovery of these elements sometimes affects multiple channels. Another goal of channel subsystem recovery design is to limit the scope of the recovery actions so that they affect as few channel programs and I/O devices as possible. The 982's design limits the scope of recovery by defining three types of recovery actions in the channels and other, more drastic recovery actions for the IOPs and channel buffer. Some of these actions involve handling passed errors. If a channel or an IOP receives an error return code from a central storage operation running on its behalf, the central storage operation keeps running and the error is passed to and recovered by the channel or IOP.

The first, most limited type of channel recovery action handles errors on the I/O interface. On the parallel interface, these errors are most commonly parity errors on the data bus. On the ESCON interface, these errors are due to bit errors on the serial link, and may be much more frequent than errors on the parallel interface. This recovery action affects only the device currently communicating on the I/O interface. If the channel detects an error, it can signal the device to attempt a command retry. If the device detects an error, it can request the channel to retry the operation. If the recovery action is successful, the channel program is not interrupted and recovery is transparent.

The second type of channel recovery deals with either internal or passed errors, which can be recovered by the channel itself without intervention by another element such as the PCE. In this case, the channel determines that there has been no damage to information describing the status of the multiple I/O operations. At most, the channel may have to terminate operation of the device logically connected to the channel at the time of the error. If an I/O operation is terminated, the channel program will see an error condition and perform the recovery operation.

The third type of recovery action also handles either internal or passed errors, but these errors damage critical information in the channel. In this case, all I/O operations terminate and the channel restarts. All the channel programs using the channel receive an error indication, and they all perform their recovery actions.

Recovery of the IOPs depends on the integrity of the channel subsystem control blocks in main storage. At most, if an IOP has an error while it is locking a control block, the other IOP or the PCE can determine which control block is locked and perform the appropriate recovery action for the affected I/O operation. If an IOP has a permanent failure, another IOP can take over the work without a loss of channels. When the channel buffer has an error, recovery involves purging all storage requests from the channels and IOPs and purging all communication buffers between the IOPs and the channels. After the purge, all the IOPs and channels are sent a signal indicating the purged condition, and it is then up to each element to perform its own recovery. This purging and notification of lost storage requests and communications is another type of passed error unique to the channel subsystem.

ESCON configuration advantages. The ESCON Multiple Image Facility (EMIF) allows channels to be shared by multiple logical partitions. Returning to Figure 3, each parallel channel is dedicated to a particular partition. The serial ESCON channels use an additional interface protocol that

allows individual channels to be shared by multiple logical partitions. This means that fewer channels are needed to provide all the necessary connections between the CPC and I/O control units. As a result, the configuration uses fewer I/O cables while maintaining redundant paths for performance and fault tolerance.

ESCON supports dynamic reconfiguration management, the ability to change the I/O configuration and describe these changes to the operating system while the system is running. I/O control units, devices, and channel paths can be added, deleted, or modified. The ability to add and delete channel paths adds more concurrent repair opportunities. For example, one can repair a trunk cable by supplying alternate paths and then removing the damaged trunk cable.

Hot pluggable channels. The 982 design allows removal and replacement of individual channel cards concurrently with continued operation of the rest of the CPC. Each channel has a dedicated interface to the rest of the channel subsystem logic; no signals are shared with any other channel. These unique interfaces allow channels to be individually isolated (or fenced) from the rest of the system. On the other hand, up to 128 channels are on the same power boundary. This presents a challenge for the power supply isolation of a single card.

Our implementation involves multiple-length card-to-board pins, which allow a card to be inserted and removed from an active board without introducing electrical noise in the remaining cards on the same board. The channels on those cards continue to execute I/O operations without interruption during the removal or replacement of the permanently failed channel card.

Figure 4 shows how the multiple-length pins gradually charge the load when a card is inserted into the board. As the card is inserted, the longest pins (numbered 1) make contact first. These pins are the ground connection. The next pins to make contact (numbered 2) supply power to the drain of an FET (field-effect transistor). The third-longest pins (numbered 3) supply the gate voltage to the FET through an RC (resistor-capacitor) network. At this time, the power supply voltage on the card is being slowly increased, preventing a large inrush of current to charge the high-capacitance load. When the card is fully inserted, the shortest pins (numbered 4) are contacted. These pins connect the board power directly to the card load. This pin length is also used for all card-to-board signal connections.

Detection of a permanent channel failure initiates an autodial for concurrent channel maintenance. Automatic maintenance procedures assure that the channel or channels on that card are taken off line. LEDs on each ESCON card allow each channel to indicate its on-line/off-line status. Every card, ESCON or parallel, also has an on-line maintenance LED, activated when the customer service engineer initiates concurrent channel maintenance. This positive visual indication

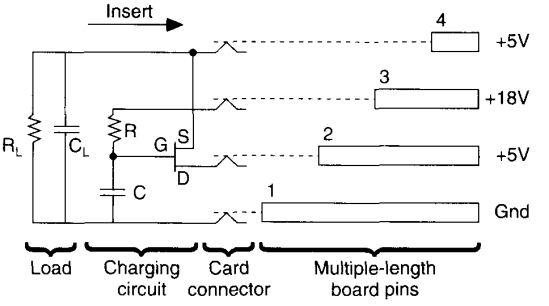


Figure 4. Hot plug charging.

assures that the correct card is replaced. By providing physical and logical isolation of individual card-on-board channels, the 982 allows channels to be repaired concurrently.

Storage hierarchy

The 982 has a four-level storage hierarchy, and fault tolerance is extensive at all levels. Level 1 is the high-speed cache of each CPU. It is divided into a 128-Kbyte data cache (level 1D) and a 128-Kbyte instruction cache (level 1I). The basic data element of both caches is a 128-byte line, and both caches are four-way set associative. Address mapping allows each line of data to be stored in any one of four sets or columns but always in a specific one of 256 rows. Level 1D is store-through cache to the next hierarchical level, the level 2 cache. Data changed by the CPU in level 1D is also held in an ECC (error correction code) protected buffer until its successful receipt by level 2.

Level 1 has parity for error detection, refetch for transient error recovery, and line delete, set delete, and relocate for hard error recovery. Line delete reduces to three the number of columns available in a particular row after a hard failure. Set delete reduces all rows to three columns. Beyond certain column thresholds, line delete is superseded by relocate, which uses built-in spare rows on the array chips to circumvent hard failures. See Spainhower et al.[12] for details of level 1 fault tolerance.

Level 2 is a store-in cache to the next hierarchical level, central storage or level 3. Level 2 has a current copy of all level 1 data. In the 982, there are two 4-Mbyte level 2s, each having data affinity with its local level 3. Level 2 has a 64-data and 8-syndrome bit, single-error correct, double-error detect ECC for checking and recovery from single-bit transient and permanent errors. For multibit permanent error recovery, level 2, which like level 1 is four-way set associative, has line and set deletes.

Central storage, also known as level 3, has a 2-Gbyte total capacity. It is protected by a 64-data and 8-syndrome bit, sin-

gle-error correct, double-error detect ECC for single-bit errors, transient or hard. Each bit of the ECC word comes from a different array chip, so all single array chip failures are correctable. Combinations of one hard and one transient or two hard failures in one ECC word are corrected by an algorithm using an exclusive-OR of complemented and recomplemented copies of the word. Spare memory chips are dynamically activated when certain single-chip thresholds that increase the likelihood of multibit alignment are exceeded. When multibit hard errors are detected, the CPC notifies the operating system to stop using the storage location. Spainhower et al.[11] includes a description of central storage fault tolerance.

Expanded storage, or level 4, is a high-speed electronic storage for frequently used data that would otherwise be stored on magnetic devices. Level 4 data is transferred to level 3 on 4-Kbyte-page boundaries. Level 4 is protected by a 128-data and 16-syndrome bit, double-error correct, triple-error detect ECC. Like level 3, it uses an exclusive-OR algorithm to correct beyond the power of its code. Also like level 3, level 4 has spare array chips.

Throughout the storage hierarchy, addresses, storage protection data, and other control information is protected by ECC, duplication, set deletes, line deletes, relocates, or some combination of these. All major data paths within storage control have ECC.

The support subsystem

The power and cooling subsystems make wide use of explicit redundancy. In the case of power supplies, fans, and blowers, all the redundant elements are normally active. The load is split across one more element—power supply, fan, or blower—than is necessary to meet physical requirements. An advantage is that the elements normally operate in a derated, less-than-full-capacity mode. This improves the reliability of many component parts. Also, regardless of which element fails, those necessary to provide uninterrupted power or cooling to functional elements are already configured and on line, eliminating any delay due to start-up or switch-over. The remaining elements simply increase their contribution to meet the needs of the load. A third advantage is that all the elements are known to be operational without latent fault diagnostics or procedures.

The power supplies, fans, and blowers can be repaired concurrently with continued operation of the 982 with no performance degradation or service outage. Visual indicators and interactive guided maintenance procedures assist service personnel in identifying elements to be replaced. For extra protection, each type of power supply has a unique connection configuration to prevent misplugging.[13]

Pumps for water cooling are also explicitly redundant but use an idle-standby scheme. To avoid situations where the active pump fails, only to switch to a backup that has a latent fault, the PCE periodically switches from the functional active pump to the idle pump. If the idle pump is not working, the PCE makes another switch-over and places a call for service. If the idle pump is functional, it changes roles with the active pump until the next scheduled switch-over.

The PCE has active and backup sides. Each side is an identical, complete PCE system with processor, memory, and disk. The backup scheme is active standby. The active side of the PCE performs all needed service and system functions for the 982. The backup side constantly runs diagnostics. The two sides are in regular communication via "heartbeat monitoring." If the backup detects itself in error, it places a service call while the active remains in control. If the active side detects itself in error, or if the backup does not receive heartbeat signals from the active, a switch-over takes place. The backup becomes the active side, and places a service call. When a side is being repaired, it is placed in a third mode, service mode. Service mode is also used to apply updates to the microcode (formally called licensed internal code).

The PCEs, IOPs, channels, and CPUs all have microcode. Service personnel usually can change the CPC's current active microcode level without any downtime. Microcode changes can be downloaded via TP link to the PCE disk at any time. The PCE is placed in service mode, and the backup side disk merges the changes with the current code to form the new microcode level. The new level is then written to the active side and the appropriate control storages within the CPC. The backup side is taken out of service mode and returned to backup mode. The one exception to this is a physically partitioned multiprocessor—for example, one 982 operating as two independent four-way CPCs. In that case, each side of the PCE is normally active and dedicated to one of the multiprocessor sides.

ON A SOLID FOUNDATION of concurrent error detection, Model 982 uses many different techniques to achieve fault tolerance. Subsystem functions mainly determine the techniques used. The support subsystem uses traditional explicit redundancy for the PCE, power supplies, water cooling, and air cooling. The implementation for each of these is different, suited to the particular subsystem component. The memory hierarchy uses a wide range of well-known techniques, including ECC and duplication, to ensure that data and control information are preserved and that the CPC continues operation even after a permanent failure. The implementation is customized for each major array in the memory hierarchy subsystem.

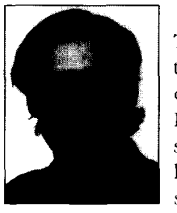
The CPUs and channel subsystem have extensive recovery mechanisms to survive transient and permanent faults. These two subsystems also achieve fault tolerance by gracefully degrading after a permanent hardware fault. The big

advantage of this design feature over more common explicit redundancy is that it permits full performance almost all the time and degrades performance only when a failure occurs. We have measured the mean time between permanent failures of a CPU, for instance, in decades. The inherent redundancy of multiple CPUs and multiple channels that exist mainly for performance also contribute greatly to the 982's fault tolerance.

References

1. D. Bossen and B. Hsiao, "Model for Transient and Permanent Error Detection and Fault Isolation Coverage," *IBM J. Research and Development*, Vol. 26, No. 1, Jan. 1982, pp. 67-75.
2. C.L. Chen et al., "Fault-Tolerance Design of the IBM Enterprise System/9000 Type 9021 Processors," *IBM J. Research and Development*, Vol. 36, No. 4, July 1992, pp. 765-779.
3. J.S. Liptay, "Design of the IBM Enterprise System/9000 High-End Processor," *IBM J. Research and Development*, Vol. 36, No. 4, July 1992, pp. 713-731.
4. IBM Corp., *Enterprise Systems Architecture/390 Principles of Operation*, Order No. SA22-7201.
5. IBM Corp., *MVS/ESA Component Diagnosis and Logic: Alternate CPU Recovery*, Order No. LY-1432.
6. J.C. Daly and F. Frey, "Nondisruptive Resuming of Work from Checkstopped Central Processor," *IBM Technical Disclosure Bulletin*, N10b, 03-92, 1991, pp. 204-205.
7. D.K. Pradhan, *Fault Tolerant Computing Theory and Techniques*, Vol. II, Prentice-Hall, Englewood Cliffs, N.J., 1986.
8. IBM Corp., *Enterprise Systems Architecture/390 IBM System/360 and System/370 I/O Interface Channel to Control Unit Original Equipment Manufacturers' Information*, Order No. GA22-6974.
9. J.C. Elliott and M.W. Sachs, "The IBM Enterprise Systems Connection (ESCON) Architecture," *IBM J. Research and Development*, Vol. 36, No. 4, July 1992, pp. 577-591.
10. J.R. Flanagan, T.A. Gregg, and D.F. Casper, "The IBM Enterprise Systems Connection (ESCON) Channel: A Versatile Building Block," Vol. 36, No. 4, July 1992, pp. 577-591.
11. C.J. Georgiou et al., "The IBM Enterprise Systems Connection (ESCON) Director: A Dynamic Switch for 200Mb/s Fiber Optic Links," *IBM J. Research and Development*, Vol. 36, No. 4, July 1992, pp. 593-616.
12. L. Spainhower et al., "Design for Fault Tolerance in ES 9000 Model 900," *Proc. Fault-Tolerant Computing Symp.*, IEEE Computer Society Press, Los Alamitos, Calif., 1992, pp. 38-47.
13. K.R. Covi, "Three-Loop Feedback Control of Fault Tolerant Power Supplies in ES/9000 Processors," *IBM J. Research and Development*, Vol. 36, No. 4, July 1992, pp. 781-785.

Lisa Spainhower is a senior engineer in the Systems Technology organization of IBM's Large Scale Computing Division in Poughkeepsie, N.Y. She is the systems availability member of several strategic product teams. She previously developed reliability, availability, and serviceability strategy for the ES/9000 Model 9021 and availability modeling for the Model 3090. She is a graduate of the University of Michigan.



Thomas A. Gregg is a senior engineer in the Connectivity Solutions organization of IBM's Large Scale Computing Division. His interests include the design of high-speed serial communications channels for large-scale computers. He contributed significantly to the development of the ESCON channel. He received an ScB in engineering and an ScM in electrical engineering from Brown University.

Ram Chillarege is the manager of the Center for Software Quality at IBM T.J. Watson Research Center. His work has focused on experimental evaluation of reliability and failure characteristics in systems. He invented orthogonal defect classification, for which he was awarded the IBM Outstanding Innovation award, and he developed the concept of failure acceleration for fault injection experiments.

Chillarege received a bachelor's degree in physics and mathematics from the University of Mysore, another in electrical engineering, and a master's degree in automation from the Indian Institute of Science. His PhD in electrical engineering is from the University of Illinois at Urbana-Champaign. He chaired the Workshop on Fault and Error Models, is an associate editor of *IEEE Transactions on Reliability*, a senior member of the IEEE, and a member of the Computer Society.

For information about this article, contact Lisa Spainhower, M/S P314, IBM, 522 South Rd., Poughkeepsie, NY 12602; or lisa@pkedvm9.vnet.ibm.com.

Reader Interest Survey

Indicate your interest in this article by circling the appropriate number on the Reader Service Card.

Low 159 Medium 160 High 161

INTRODUCTION TO COMPUTING SYSTEMS:

From Bits and Gates to C and Beyond

Yale N. Patt
The University of Texas at Austin

Sanjay J. Patel
University of Illinois at Urbana-Champaign

Boston Burr Ridge, IL Dubuque, IA Madison, WI
New York San Francisco St. Louis
Bangkok Bogotá Caracas Lisbon London Madrid Mexico City
Milan New Delhi Seoul Singapore Sydney Taipei Toronto

McGraw-Hill Higher Education
A Division of The McGraw-Hill Companies

INTRODUCTION TO COMPUTING SYSTEMS:
FROM BITS AND GATES TO C AND BEYOND

This book is printed on acid-free paper.

1 2 3 4 5 6 7 8 9 0 FGR/FGR 0 9 8 7 6 5 4 3 2 1 0

ISBN 0-07-237690-2

Vice president/Editor-in-chief: *Thomas Casson*
Executive editor: *Betsy Jones*
Development editor: *Michelle Flomenhoft*
Marketing manager: *John T. Wannemacher*
Project managers: *Laura M. Healy* and *Laura Ward Majersky*
Production supervisor: *Rose Hepburn*
Coordinator freelance design: *Matt Baldwin*
Cover design: *Andrew Curtis*
Compositor: *Techsetters, Inc.*
Typeface: *10/12 Times Roman*
Printer: *Quebecor Printing Book Company/Fairfield*

Library of Congress Cataloging-in-Publication Data

Patt, Yale N.
Introduction to computing systems : from bits and gates to C and beyond / Yale N. Patt and Sanjay J. Patel.
p. cm. – (McGraw-Hill series in computer science)
ISBN 0-07-237690-2 (alk. paper)
1. Computer science. 2. C (Computer program language) I. Patel, Sanjay J. II. Title. III. Series
QA76.P367 2001
004–dc21

00-56615

http://www.mhhe.com

chapter

7

Assembly Language

By now, you are probably a little tired of 1s and 0s and keeping track of 0001 meaning ADD and 1001 meaning NOT. Also, wouldn't it be nice if we could refer to a memory location by some meaningful symbolic name instead of memorizing its 16-bit address. And, wouldn't it be nice if we could represent each instruction in some more easily comprehensible way, instead of having to keep track of which bit of the instruction conveys which individual piece of information about the instruction. It turns out that help is on the way.

In this chapter, we introduce Assembly Language, a mechanism that does all that, and more.

7.1 ASSEMBLY LANGUAGE PROGRAMMING—MOVING UP A LEVEL

Recall the levels of transformation identified in Figure 1.6 of Chapter 1. Algorithms are transformed into programs described in some mechanical language. This mechanical language can be, as it is in Chapter 5, the machine language of a particular computer. Recall that a program is in a computer's machine language if every instruction in the program is from the ISA of that computer.

On the other hand, the mechanical language can be more user-friendly. We generally partition mechanical languages into two classes, high-level and low-level. Of the two, high-level languages are much more user-friendly. Examples are C, C++, Fortran, COBOL, Pascal, plus more than a thousand others. Instructions in a high-level language almost (but not quite) resemble statements in a natural language such

as English. High-level languages tend to be ISA independent. That is, once you learn how to program in C (or Fortran or Pascal) for one ISA, it is a small step to write programs in C (or Fortran or Pascal) for another ISA.

Before a program written in a high-level language can be executed, it must be translated into a program in the ISA of the particular computer on which it is expected to execute. It is usually the case that each statement in the high-level language specifies several instructions in the ISA of the computer. In Chapter 11, we will introduce the high-level language C, and in Chapters 12 through 19, we will show the relationship between various statements in C and their corresponding translations in LC-2 code. In this chapter, however, we will only move up a small step from the ISA we dealt with in Chapter 5.

A small step up from the ISA of a machine is that ISA's assembly language. Assembly language is a low-level language. There is no confusing an instruction in a low-level language with a statement in English. Each assembly language instruction usually specifies a single instruction in the ISA. Unlike high-level languages which are usually ISA independent, low-level languages are very ISA dependent. In fact, it is usually the case that each ISA has only one assembly language.

The purpose of assembly language is to make the programming process more user-friendly than programming in machine language (i.e., the ISA of the computer with which we are dealing), while still providing the programmer with detailed control over the individual instructions that the computer can execute. So, for example, while still retaining control over the detailed instructions the computer is to carry out, we are freed from having to remember what opcode is 0001 and what opcode is 1001, or what is being stored in memory location 0011111100001010 and what is being stored in location 0011111100000101. Assembly languages let us use mnemonic devices for opcodes, such as ADD and NOT, and let us give meaningful symbolic names to memory locations, such as SUM or PRODUCT, rather than use their 16-bit addresses. This makes it easier to differentiate which memory location is keeping track of a SUM and which memory location is keeping track of a PRODUCT.

We will see, starting in Chapter 11, that when we take the larger step of moving up to a higher level language (such as C), programming will be even more user-friendly, but we will relinquish control of exactly which detailed instructions are to be carried out in behalf of a high-level language statement.

7.2 AN ASSEMBLY LANGUAGE PROGRAM

We will describe LC-2 assembly language by means of an example. The following program multiplies a number by six by adding the number to itself six times. For example, if the number is 123, the program computes the product by adding 123 + 123 + 123 + 123 + 123 + 123.

The program consists of 20 lines of code. We have added a *line number* to each line of the program in order to be able to refer to individual lines easily. This is a common practice. These line numbers are not part of the program. Nine lines

start with a semicolon, designating that they are strictly for the benefit of the human reader. More on this momentarily. Seven lines (05, 06, 07, 0B, 0C, 0D, and 0F) specify actual instructions to be translated into instructions in the ISA of the LC-2, which will actually be carried out when the program runs. The remaining four lines (04, 11, 12, and 14) contain pseudo-ops,which are messages from the programmer to the translation program to help in the translation process. The translation program is called an *assembler* (in this case the LC-2 assembler), and the translation process is called *assembly*.

```
01  ;
02  ; Program to multiply a number by the constant 6
03  ;
04          .ORIG   x3050
05          LD      R1,SIX
06          LD      R2,NUMBER
07          AND     R3,R3,#0        ; Clear R3. It will
08                                  ; contain the product.
09  ; The inner loop
0A  ;
0B  AGAIN   ADD     R3,R3,R2
0C          ADD     R1,R1,#-1       ; R1 keeps track of
0D          BRp     AGAIN           ; the iterations
0E  ;
0F          HALT
10  ;
11  NUMBER  .BLKW   1
12  SIX     .FILL   x0006
13  ;
14          .END
```

7.2.1 Instructions

Instead of an instruction being 16 0s and 1s, as is the case in the LC-2 ISA, an instruction in assembly language consists of four parts, as shown below:

```
LABEL   OPCODE   OPERANDS   ; COMMENTS
```

Two of the parts (LABEL and COMMENTS) are optional. More on this momentarily.

Opcodes and Operands Two of the parts (OPCODE and OPERANDS) are **mandatory**. An instruction must have an OPCODE (the thing the instruction is to do), and the appropriate number of operands (the things it is supposed to do it to). Not surprisingly, this was exactly what we encountered in Chapter 5 when we studied the LC-2 ISA.

The OPCODE is a symbolic name for the opcode of the corresponding LC-2 instruction. The idea is that it is easier to remember an operation by the symbolic name ADD, AND, or LDR than by the four-bit quantity 0001,0101, or 0110. Figure 5.1 (also, Figure C.7) lists the OPCODES of the 16 LC-2 instructions. Pages 429 through 449 show the assembly language representations for the 16 LC-2 instructions.

The number of operands depends on the operation being performed. For example, the ADD instruction (line 0B) requires three operands (two sources to obtain the numbers being added, and one destination to designate where the result is to be placed). All three operands must be explicitly identified in the instruction.

```
AGAIN    ADD    R3,R3,R2
```

The operands to be added are obtained from register 2 and from register 3. The result is to be placed in register 3. We represent each of the registers 0 through 7 as R0, R2, . . . , R7.

The LD instruction (line 06) requires two operands (the memory location from which the value is to be read) and the destination register which is to contain the value after the instruction completes execution. We will see momentarily that memory locations will be given symbolic addresses called *labels*. In this case, the location from which the value is to be read is given the label *NUMBER*. The destination into which the value is to be loaded is register 2.

```
LD    R2, NUMBER
```

As we discussed in Section 5.1.6, operands can be obtained from registers, from memory, or they may be literal (i.e., immediate) values in the instruction. In the case of register operands, the registers are explicitly represented (such as R2 and R3 in line 0B). In the case of memory operands, the symbolic name of the memory location is explicitly represented (such as NUMBER in line 6 and SIX in line 05). In the case of immediate operands, the actual value is explicitly represented (such as the value 0 in line 07).

```
AND  R3, R3, #0 ; Clear R3. It will contain the product.
```

A literal value must contain a symbol identifying the representation base of the number. We use # for decimal, x for hexadecimal, and b for binary. Sometimes there is no ambiguity, such as in the case 3F0A, which is a hex number. Nonetheless, we write it as x3F0A. Sometimes there is ambiguity, such as in the case 1000. x1000 represents the decimal number 4096, b1000 represents the decimal number 8, and #1000 represents the decimal number 1000.

Labels Labels are symbolic names which are used to identify memory locations that are referred to explicitly in the program. There are two reasons for explicitly referring to a memory location.

1. The location contains the target of a branch instruction (for example, AGAIN in line 0B).
2. The location contains a value that is loaded or stored (for example, NUMBER, line 11, and SIX, line 12).

The location AGAIN is specifically referenced by the branch instruction in line 0D.

```
BRp    AGAIN
```

In the next few sections, we present the actual LC-2 code for carrying out these operations. We do so by closely examining the statement `b = NoName(a, 10);` from the code in the previous code segment.

14.6.1 The Call

In the statement `b = NoName(a, 10);`, the function **NoName** is called with two arguments. The value returned by the function is then assigned to the local integer variable **b**. In translating this function call, the compiler generates LC-2 code which does the following:

1. Transmits the value of the two arguments to the function **NoName** by writing the values in the parameter fields of a *new* activation record.
2. Stores the value of R6 into the dynamic link field of this new activation record.
3. Modifies R6 to contain the address of this new activation record. This effectively pushes a new activation record on the stack.
4. Transfers control to **NoName** via the **JSR** instruction.

The LC-2 code to perform this function call looks as follows:

```
LDR   R0, R6, #3   ; load a
STR   R0, R6, #8   ; store the first argument to NoName
AND   R0, R0, #0   ; R0 <- 0
ADD   R0, R0, #10  ; R0 <- 10
STR   R0, R6, #9   ; store the second argument to NoName

STR   R6, R6, #7   ; store R6 into the dynamic link of rec
                   ; we pushed a new record on the stack
ADD   R6, R6, #5   ; Move R6 to point to start of record.
JSR   NoName
```

The first seven LC-2 instructions accomplish the task of transmitting the argument values. When generating the code to accomplish this, the compiler needs to be aware only of the size of the activation record of the caller function. The caller function in this case is the function **main**, and it has an activation record size of five (three bookkeeping entries plus two local variables). The compiler knows that the new activation record will begin *immediately* after the activation record for **main**. It needs to know very little about the function being called aside from the number, order, and type of the parameters.

The sixth instruction (`STR R6, R6, #7`) stores the current value of R6 into the dynamic link field of the new activation record. The dynamic link is used, as we shall see, to correctly pop a record off the stack. It stores the value of the old top of stack. The seventh instruction increments R6 just beyond the current activation record and points it to the activation record for **NoName**. Finally, a **JSR** instruction initiates the execution of the callee function. Recall that the LC-2 **JSR** instruction places the return address in R7.

Figure 14.6 shows the layout in memory of these two activation records after this code has executed. Some values have not been generated, such as the return value for **main**. These values are marked with a - -.

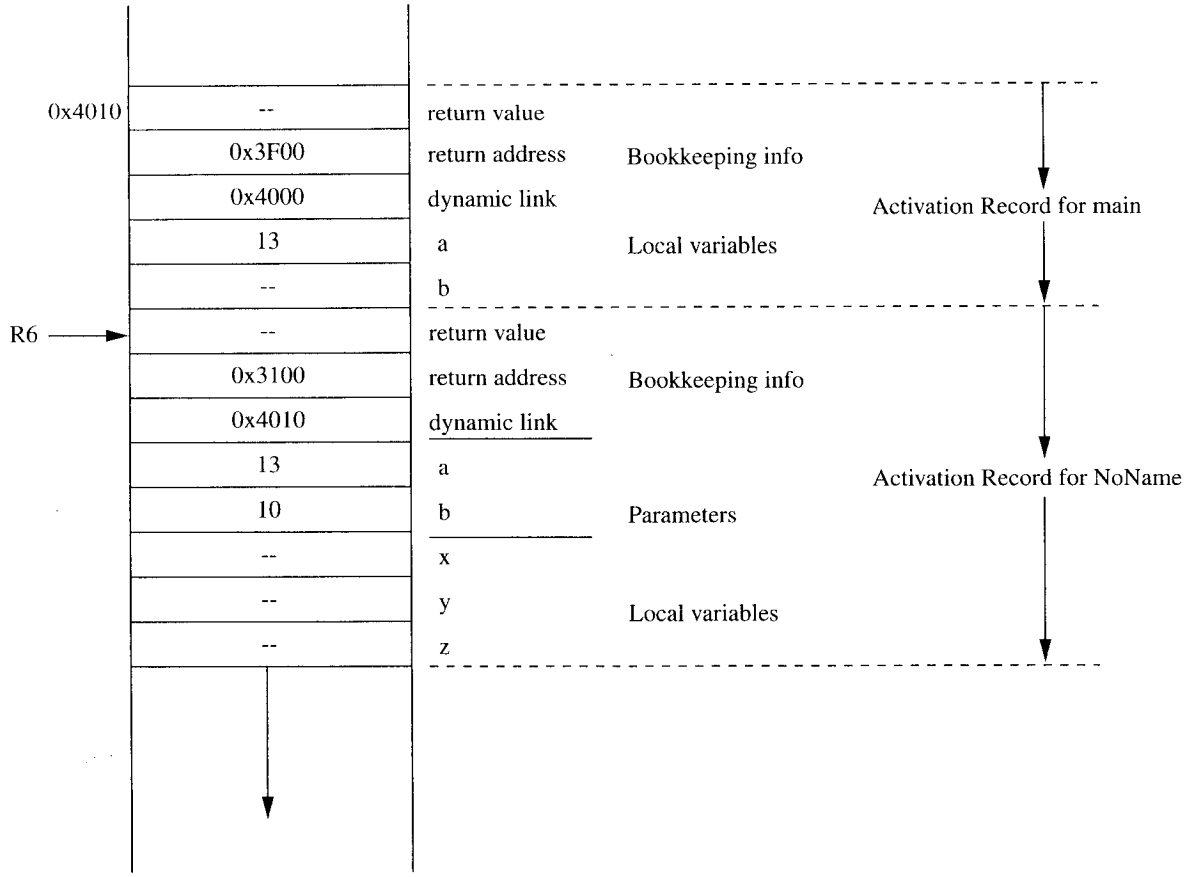


Figure 14.6 The run-time stack after the activation record for `NoName` is pushed on the stack.

14.6.2 Starting the Callee Function

The instruction executed after the `JSR` is the first instruction in the callee function `NoName`. Before the compiler can generate the code corresponding to the statements within a function, it must take care of some bookkeeping due to the function call. All functions in C (including `main`) are called from somewhere, thus all functions must start off by saving the value of R7 (it contains the return address of where they need to return control) into the return address entry in the activation record. The LC-2 code to do this is quite simple:

```
NoName:
        STR R7, R6, #1    ; store return address
```

When the `JSR` in the caller function executes, the address of the instruction following the `JSR` in the code is put into R7. The first task of the callee function is to store the value in a safe place in memory—losing the return address would leave no way to return control to the caller function. For this reason, each activation record contains a spot where we can safely store the return address.

Combining RISC and CISC in PC systems

Simon C.J. Garth

Abstract

RISC processors are being increasingly used in computers in which performance is a key feature. However, their take-up may be restricted by the limited availability of software compared to that available for established machines, normally based on CISC processors. This paper describes a method in which RISC and CISC processors have been combined into the same machine in order to offer compatibility with a wide software base (that of the PC) and, at the same time, access to state-of-the-art computing performance. The hardware and software issues involved and the level of integration which can be achieved are also considered.

Introduction

By far the most popular computers are those based on the architecture of the IBM PC. They offer a wide range of performance and the largest software base of any such machine. This has been achieved, at least in part, by preserving compatibility as new, more powerful models are introduced.

The need for compatibility with previous machines frequently delays the introduction of new improved architectures of both the computer and the underlying central processing unit (CPU). This has been particularly true of machines based on CISC technology (such as the PC) and frequently results in computers made from leading edge components suffering from a lack of application software and computers with the largest software base being based on technology which is less than state of the art.

One method of confronting this dilemma is to incorporate more than one CPU of different design into a single computer. This allows software designed for different CPU architectures to be run on their respective processors within the same machine, giving the appearance of a 'multi-lingual' computer.

System architecture

The CISC technology in the PC is based on the Intel 80x86 architecture. Early members of this family used segmented memory, a restricted number of internal registers and external floating point support. Newer members have alleviated some of these limitations but not to the degree which may be achieved in some RISC processors. The addition of one or more additional (or 'attached') processors allows the performance of a RISC processor to be combined with the compatibility of a CISC processor.

Simon Garth is Technical Director of Myriad Solutions Ltd.

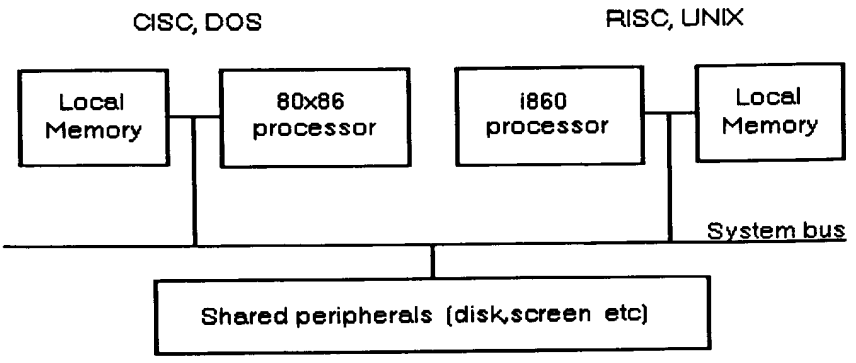


Fig. 1 System architecture

The Intel i860 was chosen as the attached processor because of its leading edge numerics capability and its ability to support standard operating systems. This device has an advanced 64 bit architecture and an instruction set which is capable execution rates on suitable code in the order of 10 times that of the highest specification 80x86 processor (for instance floating point operations). The attached processor subsystem may be regarded as a main processor board in its own right with local memory and a restricted set of peripherals (e.g. interrupt handler and timer). It plugs into the expansion bus of the host and shares the host's I/O devices.

The coupled two-processor architecture allows the 80x86 to run code designed for it (i.e. conventional code for the PC) but other programs which requires the high speed of the attached processor may be ported to that processor. The port may involve re-compilation of existing source code or the use of applications which have been specifically designed to use the features of the RISC processor. The processors may run on independent tasks or co-operate in a tightly coupled mode with a single application split over both processors. An example of such a case would be an engineering simulation program in which the graphical front end runs under the host operating system's Graphical User Interface (such as X-Windows or MS-Windows) while the numerically intensive routines are executed in the attached processor. This provides the user with familiar displays and means of interaction with the product and, in addition, the performance benefits of the attached processor.

System integration

The computational power afforded by such an attached processor far outstrips that available on conventional PCs (or most workstations). The major issue remaining is the means by which this processing power may be harnessed. This involves consideration of a number of areas including operating systems, access to I/O devices and compatibility with existing code.

UNIX is the operating system most commonly associated with RISC-based computers and offers many powerful functions. When combining RISC and CISC it is attractive to combine the benefits of the UNIX operating system on the RISC processor with the familiarity and

compatibility of MS-DOS on the host. This allows users to run applications on the RISC processor which take advantage of the features of UNIX (flat memory model, demand paging, virtual memory management) but without the overhead of a full UNIX system.

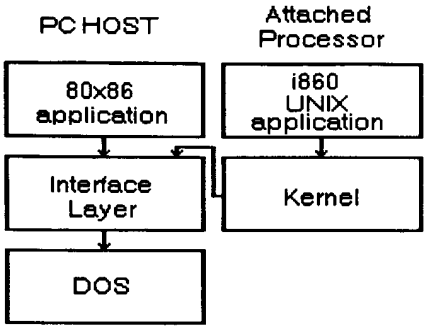


Fig.2 Software integration across the two processors

In the example being discussed here, the integration of DOS and UNIX is achieved by adding a UNIX-like kernel to the RISC processor and providing an interface layer in software which links the two operating systems and hence the two processors. The principal function of this interface is to convert system calls which are not supported in the kernel (such as requests to access I/O devices) into the equivalent system calls on the host. In this way, the RISC processor appears to the application software to have direct access to all the host's peripherals.

Applications which are to be run on the attached processor have added to them a small 'stub' program which is executed first when the application is invoked and the function of which is to load the main body of the program software to the attached processor and initiate execution. This occurs transparently to the user and results in a machine which appears to be 'bilingual', being able to run either code designed for the host or the attached processor in a seamless manner. This concept may be extended, using a multi-tasking operating system or graphical user interface on the host, to a machine which runs DOS and UNIX applications simultaneously on the respective processors and which is able to share data (e.g. via disk files) and resources between the applications.

Further extensions are possible towards systems containing multiple attached processors, each running one or more applications. This allows the possibility when adding new software, to add hardware which is optimised for running that software. This is particularly critical in real time applications in which the total quantity of software which may be run simultaneously is restricted by the power of the computer. By adding hardware with software, the ability of the computer to run existing applications is not significantly impacted.

The i860

The i860 follows many of the characteristics of RISC processors including single cycle instruction execution, and single load/store operations. The architecture may be described as a

series of semi-autonomous processing units with a high bandwidth datapath between them. The core unit is responsible for all integer and logical operations as well as load/store initiation and branch operations. There are separate floating point multiply and adder units which are able to operate independently and in parallel both with each other and with the other units. In addition, there is a dedicated graphics unit providing functions in hardware to assist with 3-D drawing operations. These units combine to give a peak performance of 50 Million integer instructions (MIPS) concurrently with 100 Million Floating Point Operations per second (MFLOPS).

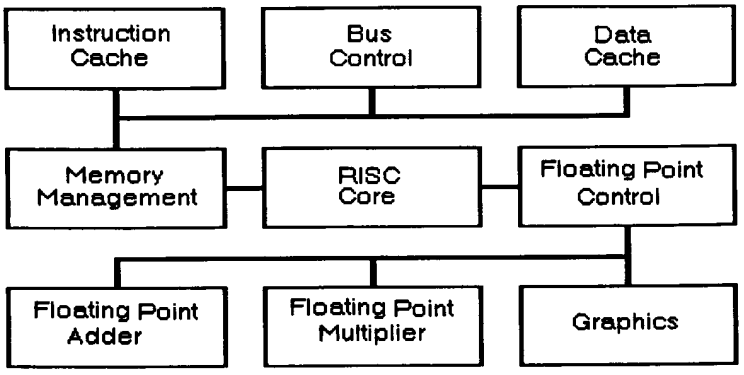


Fig.3 i860 functional units

One of the major problems in integrating applications with for such high performance processors is the requirement to keep them fed with data. In the worst case, it is necessary to access two 64-bit data terms and one 32 bit instruction per clock. Even with a 64 bit external data bus, it is not feasible to access three arbitrary terms within one clock cycle. The solution to this problem is provided by a number of architectural features on the chip including a very large register file (64 32-bit registers), separate instruction and data caches and a number of external memory access modes.

The data bandwidth from the cache is over 1GBit/sec. This is sufficient to satisfy the above requirements if all the data is resident in cache. In addition, intermediate terms in computations will frequently reside in registers which have their own busses within the processor. Many intensive operations are performed in loops which can usually be contained in the instruction cache, alleviating the need for external instruction fetches. Nevertheless there is still a severe load placed on the external bus.

The i860 provides a number of mechanisms to make external access to main memory efficient. The processor is capable of external memory pipelining, in which it may issue requests for new data terms before previous data accesses have completed. In addition, the i860's memory manager uses the caches to allow it to fetch several instructions or items of data in blocks and to store them in the caches until required. This is augmented by a burst transfer mode across the external bus which is capable of transferring one 64 bit term per clock (i.e. a data rate of 400MBytes/sec).

The predominant type of memory used in computer systems is Dynamic Random Access Memory (DRAM). These memories are organised in such a way that accesses to addresses which are located within a 'page' (typically 1kBytes in size) may be fetched very much faster than accesses whose addresses are in separate pages. By grouping memory accesses in this way, the time taken to access an individual item is reduced and the external bus is able to operate at close to its peak bandwidth.

Register scoreboarding is implemented which allows instructions to be issued every clock and proceed as far as possible until a 'freeze condition' occurs because the data on which they must operate is invalid. The combination of register scoreboarding and pipelining allow the performance of the system to be optimised, even with relatively slow external memory.

Consider an example in which the processor is to add the elements of an array of floating point numbers. The code will consist of the following operations:

```
.
Fetch next term to add
Perform addition
Test for end of array
If not end, branch to beginning of loop
.
```

In a conventional processor, there would be delays between the fetch of data and add instruction and between the add instruction and test. An optimising compiler for the i860 would re-schedule the instructions as follows:

```
.
Fetch next term to add
Test for end of array
If not end, branch to beginning of loop
(Perform addition)
.
```

The test operation will follow in the next clock to the fetch instruction as it does not depend on the data being fetched. The addition has been placed in the delay slot of a delayed branch and will execute whether or not the branch is taken. In addition, if the addition takes more than one clock to execute, the next fetch instruction may be initiated before the addition is complete. In this manner, the integer, floating point and memory manager units of the processor are able to operate in parallel giving many times the performance of earlier generation processors, the instructions of which were executed strictly sequentially.

Conclusion

Attached processors may be used to combine the capabilities of leading-edge RISC processors and their associated operating systems with the familiarity and large software base of more conventional computers. The resulting ability to augment computational power by adding extra hardware when software is added provides an effective means of accommodating new, more demanding, applications within the confines of an existing machine.

An established microprocessor family—especially its newest member—offers speed, compatibility with a large body of existing software, and multiprocessing capability.

An Overview of the 9900 Microprocessor Family

Richard V. Orlando
Thomas L. Anderson
American Microsystems, Inc.

A recent article in *IEEE Micro* (Toong and Gupta, May 1981[1]) discussed at some length the main features of several currently available 16-bit microprocessors. Although the authors made no pretense about covering all such processors, we feel they erred in omitting at least one machine, the 9900. As we will show, the 9900 microprocessor and its successor, the 9995, are powerful machines with architectural and performance characteristics rivalling those of the 8086, Z8000, and 68000. We will also discuss the recently disclosed 99000 family, a highly sophisticated, upward-compatible extension to the 9900 family.

Besides its power, the 9900 is notable for being the first commercially available 16-bit microprocessor—Texas Instruments first offered the TMS9900 in 1976 and it is still on the market. In addition, the 9900 is second-sourced by AMI as the S9900. We will briefly outline the salient features of this processor and its successors in a manner similar to that used by Toong and Gupta—by following the format of the original article, we hope to present a valid comparison to the machines discussed there.

General characteristics

Since the 9900 was the pioneering 16-bit microprocessor, it does not contain all the features found on later machines. However, it does provide 16-bit internal and external data buses, byte and word instructions, and most of the common addressing modes. The machine architecture provides 16 general-purpose registers, although these actually reside in main memory. This strategy achieves one of the benefits of register addressing: the coding efficiency gained by not having to specify full memory addresses. However, no increase in speed is attained by using register instead of memory instructions. The address space is 64K, with no internal provision for extension. The 9900 emphasizes the "family" approach to microprocessors, with several different software-compatible models available. The 9940 is a single-chip microcomputer with an enhanced 9900 as its CPU; the 9980 is an 8-bit data bus version of the 9900; the 9995 is a high-performance version of the 9980. We will examine the 9900 and the 9995, pointing out the differences between them where appropriate.

Architectural details

The basic structure of the 9900 is shown in Figure 1; its specifications are listed in Table 1 (along with those for the 9995). The operation of the processor is straightforward. There is no instruction prefetch or pipelining. The I/O interface is handled by a communications register unit, or CRU, which uses specific instructions to address external devices via the CRU lines and address bus. Most important, I/O bits can be addressed individually or in fields of 1 to 16 bits. The structure of the 9995 (Figure 2) is similar to that of the 9900, although it adds a single instruction prefetch, 256 bytes of internal RAM, and internal clock generation. To save pins and to allow the use of byte-wide memories, the 9995 provides only an eight-bit external data bus. However, the 16-bit-wide on-chip RAM prevents this from becoming a bottleneck.

Register organization. 9900 processors contain three primary internal registers: the program counter (PC), the status register, and the workspace pointer (WP). The

0272-1732/81/0800-0038$00.75

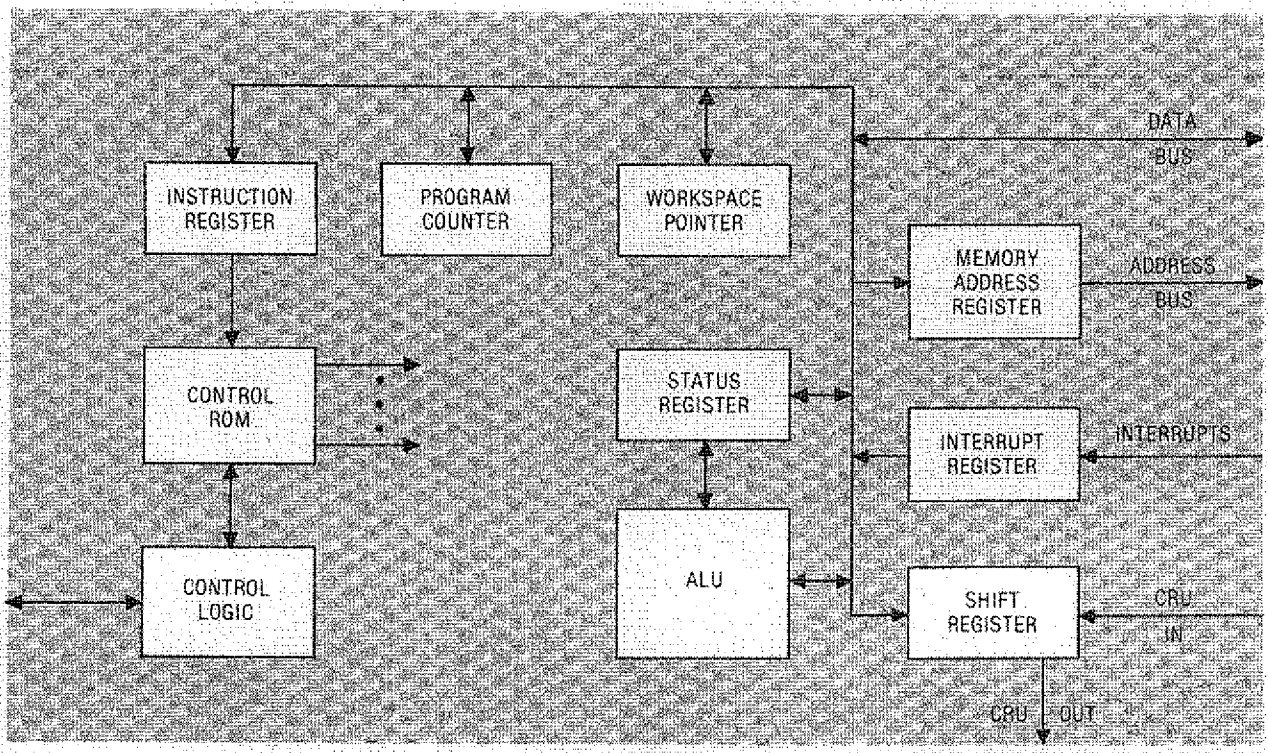


Figure 1. Basic structure of the 9900.

status register contains a four-bit interrupt mask and processor status bits (seven for the 9900 and eight for the 9995). The WP points to the starting location in memory of the 16 general-purpose "workspace" registers, which must be held in contiguous words. A "context switch" mechanism loads the WP with a new value, thus defining a new set of work-space registers in main memory. This multiple-register-set scheme allows the 9900 family to efficiently handle interrupt and subroutine calls without saving or stacking the contents of the workspace registers. In essence, the 9900 family uses a linked-list approach to program control linkage rather than the more traditional hardware stack. Since certain workspace locations are used to store processor status during subroutine calls, the "general-purpose" registers are not completely general. The 9995's 256 bytes of internal RAM are commonly used to store workspaces—this greatly improves the performance of register operations.

System structure. The microprocessor, memory, and I/O devices are interconnected by a local bus as defined by the processor pin-out. The data portion of this bus is used only for memory operations; I/O is handled by a separate serial interface (described in a later section). A microprocessor, memory, and I/O devices form a module which may be connected through bus control and arbiter modules to a global bus (see Hirschman, Ali, and Swan[5]).

Memory. The 9900 and 9995 use a straightforward 64K address space, which can be expanded only by external memory management. The 9900 always accesses an entire 16-bit word of memory at one time, although it has byte instructions to perform operations on either of the two bytes.

Stack organization. The 9900 and 9995 are not stack-oriented machines and provide no direct hardware support for stacks. For example, for a subroutine call or interrupt response, several of the work-space registers—rather than a stack—are used to save the processor state. However, it is a relatively straightforward matter to set up a stack of workspaces to permit recursive and multiple subroutine calls.

I/O mechanisms. Up to 4K bits each of input and output may be individually addressed by the CRU mechanism in the 9900, and up to 32K bits in the 9995. These bits may be addressed in fields of 1 to 16 bits, which allows single-operation flag testing or setting. Data transfer to and from I/O devices is handled serially through the CRUIN and CRUOUT lines. External decoding of the low-order address lines determines the actual number of I/O devices being addressed.

Software. Unlike other 16-bit microprocessors, the 9900 was designed to be software-compatible with a minicomputer family—the Texas Instruments 990 series—rather than with a microprocessor. Texas Instruments' philosophy of microprocessor design dictates that advances in minicomputer systems development be incorporated into new microprocessors as soon as improvements in VLSI technology permit it. The result is a parallel development of both a microprocessor and a minicomputer family, with software compatibility between families and among individual members of each family.

The 9900 family supports a full memory-to-memory architecture, although workspace registers are provided to reduce program size through encoding efficiency. The in-

struction set supports most addressing modes, some byte instructions, and bit-addressable I/O space. The 9995 contains arithmetic overflow traps to help handle run-time errors. The 9900 and 9995 both contain software interrupts, which allow a user to emulate a new instruction in macrocode. The 9995 also provides an unimplemented instruction trap (MID interrupt) to allow users to simulate complex instructions or to trap on illegal opcodes.

The 99000. Texas Instruments recently disclosed the 99000 family of 16-bit microprocessors.[6,7] Three members of this family are under development—the 99105 is a faster version of the 9900; the 99110 and 99120 (Figure 3), however, provide numerous enhancements to the 9900 family which greatly increase their range of application. Each has single instruction prefetch, an internal oscillator and clock generator, arithmetic overflow and illegal opcode traps, and status output pins for multiprocessor and DMA configurations. The instruction set includes many extensions to the 9900, including long-word arithmetic, support for user-defined stacks, and test-and-set primitives for semaphores. The 64K address space may be extended to 16M by using the TIM99610 memory manager; the 99110 and 99120 include instructions to support this chip.

These processors also include a feature called the macrostore, an internal high-speed memory addressed independently of main memory and currently comprising 1K bytes of ROM and 32 bytes of RAM. This fast memory allows the software emulation of new instructions or frequently executed routines to operate considerably faster than if the instructions or routines were stored in main memory. The 99110 will contain floating-point routines as part of its macrostore; the 99120 will contain the kernel of TI's Real-Time Executive, which will support a Pascal-based operating system. These new architectural features, when coupled with the increase in operating frequency, will yield significant improvements over 9900 family execution times. In addition, the 99110 and 99120 can operate in either user or supervisor (privileged) mode.

Table 1.
Specifications for the 9900 family.

	9900	9995
YEAR OF COMMERCIAL INTRODUCTION	1976	1981
NO. OF BASIC INSTRUCTIONS	69	73
NO. OF GENERAL-PURPOSE REGISTERS	16	16
PIN COUNT	64	40
DIRECT ADDRESS RANGE (BYTES)	64K	64K
NUMBER OF ADDRESSING MODES	8	8
BASIC CLOCK FREQUENCY	3 MHz*	3 MHz
SYSTEM STRUCTURES		
UNIFORM ADDRESSABILITY		
MODULE MAP AND MODULES		
VIRTUAL		
PRIMITIVE DATA TYPES		
BITS	•	•
INTEGER BYTE OR WORD	•	•
INTEGER DOUBLE-WORD		
LOGICAL BYTE OR WORD	•	•
LOGICAL DOUBLE-WORD		
CHARACTER STRINGS (BYTE, WORD)	•	•
CHARACTER STRINGS (DOUBLE-WORD)		
BCD BYTE		
BCD WORD		
BCD DOUBLE-WORD		
FLOATING-POINT		
DATA STRUCTURES		
STACKS	•	•
ARRAYS	•	•
PACKED ARRAYS	•	•
RECORDS	•	•
PACKED RECORDS		
STRINGS		
PRIMITIVE CONTROL OPERATIONS		
CONDITION CODE PRIMITIVES	•	•
JUMP	•	•
CONDITIONAL BRANCH	•	•
SIMPLE ITERATIVE LOOP CONTROL	•	•
SUBROUTINE CALL	•	•
MULTIWAY BRANCH		
CONTROL STRUCTURE		
EXTERNAL PROCEDURE CALL		
SEMAPHORES		
TRAPS	•	•
INTERRUPTS	•	•
SUPERVISOR CALL	•	•
OTHERS		
USER MICROCODE		
DEBUG MODE		

*The 9900 clock frequency is 3.0 MHz, but since two clock cycles are required for each machine state, the effective clock frequency is actually only 1.5 MHz.

Microcomputers

There are several single-card microcomputers based on members of the 9900 family. Table 2 summarizes the characteristics of the Texas Instruments TM990/101 single-card microcomputer, which contains a 9900 as its CPU.

Multiprocessor capabilities

As already mentioned, the designer can configure 9900 family processors in a multiprocessor system by interconnecting them on a global bus. Although no *specific* multiprocessing features were incorporated into the design of the 9900 and 9995, the 99110 and 99120 provide several features designed specifically for multiprocessing environments. They send out bus status codes so that other modules in the system know exactly what phase of instruction execution they are in. Such status information is critical to efficient arbitration of system bus contention. They provide primitives for testing and setting semaphores, with external signals to lock out other processors during atomic operations.

Selection strategy

Technical issues. The machines of the 9900 family have both advantages and disadvantages when compared to the 8086, Z8000, 68000, and NS16000. Their direct address space, 64K, is the smallest of the group, although this address space is externally extensible. Their I/O facilities are addressed separately from memory, allowing implementation of useful features such as individual bit access.

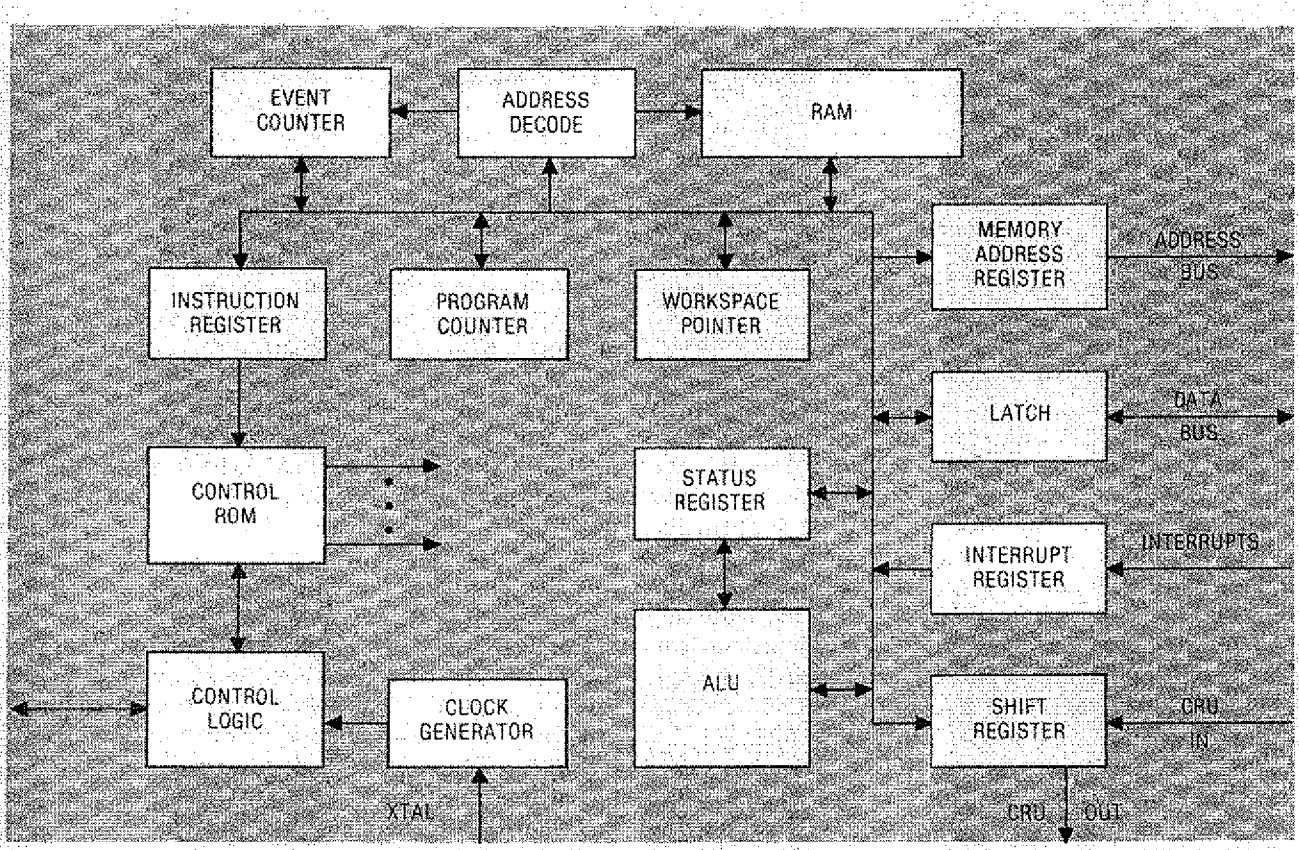


Figure 2. Basic structure of the 9995.

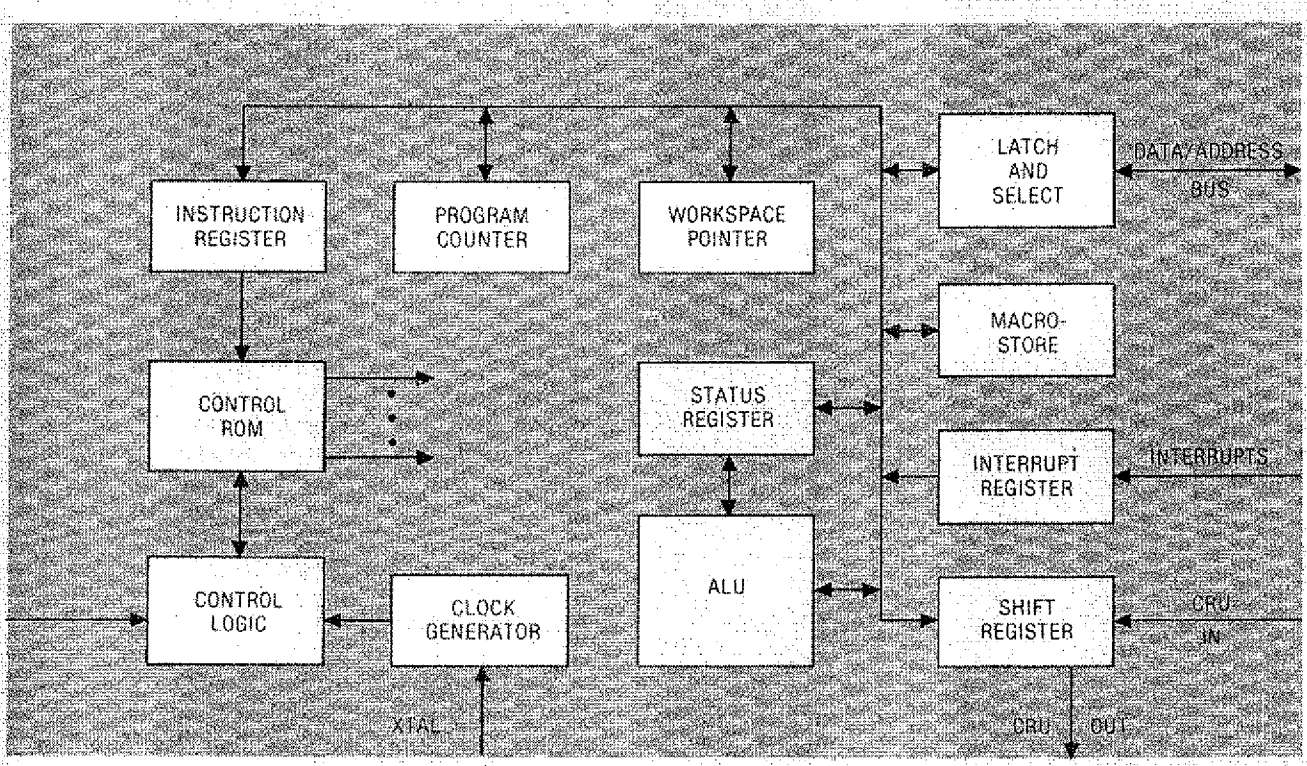


Figure 3. Basic structure of the 99110 and 99120.

Unlike the Z8000 and 68000, currently available members of the 9900 family do not support 32-bit operands, except in multiplies and divides. However, the family's memory-to-memory architecture provides an advantage not found in the other 16-bit micros by reducing the number of instructions required to access operands. The 99110 and 99120 support both long-word operations and a memory-to-memory architecture; hence, their performance benefits greatly. Table 3 lists the execution speeds of various instructions for the 3-MHz (1.5-MHz effective) 9900, the 3-MHz 9995, and the 6-MHz 99110.

The 9900 is obviously the slowest currently available 16-bit microprocessor, although it is the least expensive and most mature. Advances in VLSI technology and microprocessor architecture are responsible for the dramatic increases in the execution speeds provided by the newer processors. The 9900 and the 9995 exemplify this—they show great differences in performance even though they both operate at the same cycle time and with the same general internal architecture. This increase can be directly attributed to architectural techniques such as instruction prefetch. The performance of the 99110 is impressive, consistent with its high clock frequency and advanced architectural features.

Applications. The ideal "application niche" for the 9900 and 9995 is probably a control environment. Because such applications do not require large amounts of memory, the limited direct address space of the 9900 and 9995 is not a major factor. The 9900 family's fast interrupt response time (as can be seen from the figures in Table 3 for context switching and restoring) is a useful feature in such environments. Its ability to individually address I/O bits and fields without the need for masks makes it particularly useful in bit-map applications such as terminals and printers. On the other hand, the serial I/O is not particularly well-suited for applications which require large amounts of data transfer.

Table 2.
TM990/101 microcomputer characteristics.

GENERAL	
PROCESSOR USED	TMS9900
WORD SIZE (BITS)	16
ADDRESSING	
ADDRESS SIZE (BITS)	15
TOTAL MEMORY ADDRESSABLE (BYTES)	64K
AMT. OF RAM ON CARD (BYTES)	4K
AMT. OF ROM ON CARD (BYTES)	0-8K
DMA CAPABILITY	YES
FREQUENCY, ETC.	
CLOCK FREQUENCY (MHz)	3
SUPPLY VOLTAGES	+5, +12, −12
BOARD SIZE (INCHES)	7.5 × 11
I/O CAPABILITY	
BUS TYPE	SPECIAL
PARALLEL I/O LINES	16+
NUMBER OF I/O PORTS	2
MAX. I/O RATE (K BAUD)	38.4
ADDITIONAL HARDWARE DETAILS	
INTERRUPT PROVISIONS	YES
MULTIPROCESSING CAPABILITY	YES
NO. OF TIMERS	3
BITS PER TIMER	8-14
SOFTWARE	
OPERATING SYSTEM	YES
HIGH-LEVEL LANGUAGE(S)	YES
ASSEMBLER	YES
DEBUGGING AIDS	YES
APPLICATION PACKAGES	YES

An important feature of the 9900 family is its provision for single-chip, 16-bit microcomputers. The 9940, for example, contains 2K of ROM, 128 bytes of RAM, internal clock generation, and 32 bits of general-purpose I/O ports in a single, 40-pin package. The 9995 includes internal clock generation and 256 bytes of on-chip RAM, making it somewhat of a hybrid between a microcomputer and a microprocessor. The availability of microcomputers is important in many control applications, where space and speed constraints do not permit use of board-level systems.

The 99110 and 99120 have enough enhancements that their usefulness will extend well beyond control applications. The two devices provide multiprocessor and multiuser support of the same caliber as that found in the Z8000 and 68000. They do not provide support for demand paging, which may limit their use in certain large applications. However, they do support functional paging—e.g., of separate data and program memories—and this can be useful in numerous applications.

There are many support chips for the 9900 family (Table 4). Although the 9900 does not have a set of peripheral chips from an eight-bit predecessor to fall back on, it has been around longer than the other 16-bit microprocessors discussed. Thus, there has been sufficient time to develop support chips for typical applications.

Commercial issues. The 9900 is second-sourced by AMI and, internationally, by ITT Intermettel. It is a mature product with the largest established software and hardware base in the 16-bit world. Both Texas Instruments and AMI will continue to support the family, as well as develop its future generations.

We have attempted to evaluate the 9900 family using the same metrics as those used by Toong and Gupta in their evaluation of 16-bit microprocessors. We conclude by presenting our own rating of the 9900, 9995, and 99000 family (Table 5) and include for comparison the ratings assigned by Toong and Gupta to the 8086, Z8000, MC68000, and NS16000. Our results show performance impressive enough to demand the 9900's inclusion in any treatment of currently available 16-bit machines. ■

Acknowledgments

The assistance of Dave Laffitte, John Schabowski, and others in the 16-bit microprocessor group at Texas Instruments was invaluable. Their proofreading and assistance in obtaining 99000 information contributed greatly to this article.

Table 3.
Execution speeds (in microseconds) of 9900 family microprocessors.

OPERATION	DATA TYPE	9900	9995	99110
REGISTER-TO-REGISTER MOVE	BYTE/WORD	4.60	(1.30)*	0.50
	DOUBLE-WORD	9.80	(2.60)	1.00
MEMORY-TO-REGISTER MOVE	BYTE	7.30	(1.99)	0.83
	WORD	7.30	(2.33)	0.67
	DOUBLE-WORD	14.60	(4.66)	1.33
MEMORY-TO-MEMORY MOVE	BYTE	9.90	(1.30),2.60	1.00
	WORD	9.90	(1.90),3.30	0.83
	DOUBLE-WORD	19.80	(3.90),6.60	1.67
ADD MEMORY TO REGISTER	BYTE/WORD	7.32	(2.60)	0.83
	DOUBLE-WORD	21.30	(5.30)	2.00
COMPARE MEMORY TO MEMORY	BYTE/WORD	9.90	1.99	1.00
	DOUBLE-WORD	19.80	3.98	2.00
MULTIPLY MEMORY-TO-MEMORY	BYTE	21.90	(7.90),8.60	4.17
	WORD	21.90	(7.90),8.60	4.17
	DOUBLE-WORD	180.64	59.95	26.38
CONDITIONAL BRANCH	BRANCH TAKEN	3.60	1.30	0.50
	BRANCH NOT TAKEN	2.90	1.30	0.50
MODIFY INDEX BRANCH IF ZERO	BRANCH TAKEN	7.60	2.60	1.00
BRANCH TO SUBROUTINE		7.90	3.90	1.00
CONTEXT SWITCH		13.60	5.60	2.00
RETURN CONTEXT		5.90	2.30	1.00

*Times in parentheses are for references to internal RAM.

Table 4.
9900 family support chips.

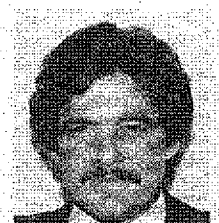
MEMORY MANAGEMENT	TIM99610
BUS ARBITER	
FLOATING-POINT	
DMA CONTROLLER	TMS9911
INTERRUPT CONTROL UNIT	TMS9901
I/O PROCESSOR/INTERFACE	TMS9901
PERIPHERAL CONTROLLER	TMS9901
FLOPPY DISK CONTROLLER	TMS9909
CRT CONTROLLER	TMS9927
ARRAY PROCESSOR	
BUBBLE MEMORY CONTROLLER	

Table 5.
Ranking of 16-bit microprocessors.

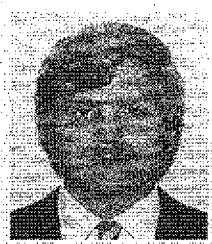
	8086	Z8000	MC68000	NS16000	9900	9995	99000
SPEED	C	B	A	A	D	B	A
NUMBER OF REGISTERS	B	A	A	C	A	A	A
ADDRESS RANGE	D	A	A	B	D	D	D
COMPATIBILITY WITH EARLIER MICROPROCESSORS	A	B	B	B	NA	A	A
SUPPORT CHIPS	A	B	C	D	A	A	A
MULTIPROCESSING CAPABILITY	B	B	B	C	C	C	A
SECOND SOURCE	A	A	A	D	B	D	D

References

1. Hoo-min D. Toong and Amar Gupta, "An Architectural Comparison of Contemporary 16-Bit Microprocessors," *IEEE Micro*, Vol. 1, No. 2, May 1981, pp. 26-37.
2. Henry A. Davis, "Comparing Architectures of Three 16-Bit Microprocessors," *Computer Design*, Vol. 18, No. 7, July 1979, pp. 91-100.
3. *9900 Family Systems Design and Data Book*, Texas Instruments, Inc., Houston, 1978.
4. *TMS 9995 Microcomputer Preliminary Data Manual*, Texas Instruments, Inc., Houston, 1981.
5. Alan D. Hirschman, Gamil Ali, and Richard Swan, "Standard Modules Offer Flexible Multiprocessor System Design," *Computer Design*, Vol. 18, No. 5, May 1979, pp. 181-189.
6. David S. Laffitte and Karl M. Guttag, "Fast On-Chip Memory Extends 16-Bit Family's Reach," *Electronics*, Vol. 54, No. 4, Feb. 24, 1981, pp. 157-161.
7. David Laffitte, "New-Generation 16-Bit Microprocessors—Fast and Function-Oriented," *Electronic Design*, Vol. 29, No. 4, Feb. 19, 1981, pp. 111-117.



Rick Orlando is manager of the Microprocessor/Microcomputer Applications Group at American Microsystems, Inc., of Santa Clara, California. His research interests include processor architecture, multiprocessing, networking, and computer graphics hardware development. A member of Tau Beta Pi, Eta Kappa Nu, the IEEE, and the IEEE Computer Society, Orlando received a BS in computer systems engineering from the University of Massachusetts at Amherst in 1980.



Thomas L. Anderson is employed for the summer of 1981 as an applications engineer for American Microsystems, Inc., of Santa Clara, California. Currently he is pursuing an SM in computer science and engineering at MIT, having received his BS in computer systems engineering from the University of Massachusetts at Amherst in 1980. His research interests include computer architecture, multiprocessor systems, and microprocessor applications. Anderson has previously worked for IBM and Digital Equipment Corporation and is a student member of Tau Beta Pi, Eta Kappa Nu, ACM, SIAM, and the IEEE.

The authors' address is American Microsystems, Inc., 3800 Homestead Road, Santa Clara, CA 95051.

FEATURE

A Floating-Point VLSI Chip for the TRON Architecture:

An Architecture for Reliable Numerical Programming

Preliminary evaluations of this FPU show a good combination of precision and performance. It supports IEEE floating-point arithmetic and the ANSI C draft proposal.

Shumpei Kawasaki
Mitsuru Watabe
Shigeki Morinaga

Hitachi Ltd.

The TRON project[1] advocates a network system consisting of heterogeneous computing nodes called intelligent objects that have a wide range of functions. Figure 1 illustrates how a standard network connects the intelligent objects so that they can communicate with each other. The TRON architecture also specifies a processing element to be used as a component of the intelligent objects for the VLSI CPU,[2] a standard instruction set for a central processing unit. Multiple vendors will provide the Gmicro microprocessor family,[3] a series of VLSI (very large scale integration) chips conforming to the architecture. Among these microprocessors are the Gmicro/100, Gmicro/200, and Gmicro/300.

While the TRON architecture for the VLSI CPU handles integer, bit-field, string, list-structure, and bitmapped data, it excludes the floating-point data intended to be handled by the coprocessor or library. As some intelligent objects will undoubtedly handle real number data, floating-point instructions are urgently needed in the architecture for the VLSI CPU. With this in mind, we designed the Gmicro/FPU to provide floating-point instructions for both the Gmicro/200 and the Gmicro/300. The VLSI CPU architecture defines 23 coprocessor instructions, some of which are designed to be used in the floating-point instructions.

Since the TRON project is meant to provide information infrastructure for various layers of society, the reliability of its parts is the key issue. The intelligent objects can participate in navigation, construction, manufacturing, chemical-plant control, air-traffic control, and even exploration of the universe. Therefore, their reliability can affect the lives of individuals and the public in general. As these applications massively utilize floating-point numbers, the numerical integrity of the Gmicro/FPU potentially could become a social issue. Thus, though performance is an important criterion in the development of the Gmicro/FPU, accurate data are even more important.[4]

Background

A floating-point number is an attempt to express real number data in digital information. A conventional floating-point number consists of a sign bit, exponent field, and mantissa field as seen in Figure 2.

0272-1732/89/0600-0026$01.00 © 1989 IEEE

The value of a floating-point number is determined by:

$$\text{value} = (-1)^s * 2^{e-\text{bias}} * m$$

where s is the sign bit, e is the exponent, and m is the mantissa. Bias is a constant that enables the floating-point number to express a value larger or smaller than one. A wider exponent field would increase the range of the value expressed by the format. A wider mantissa field would increase its resolution.

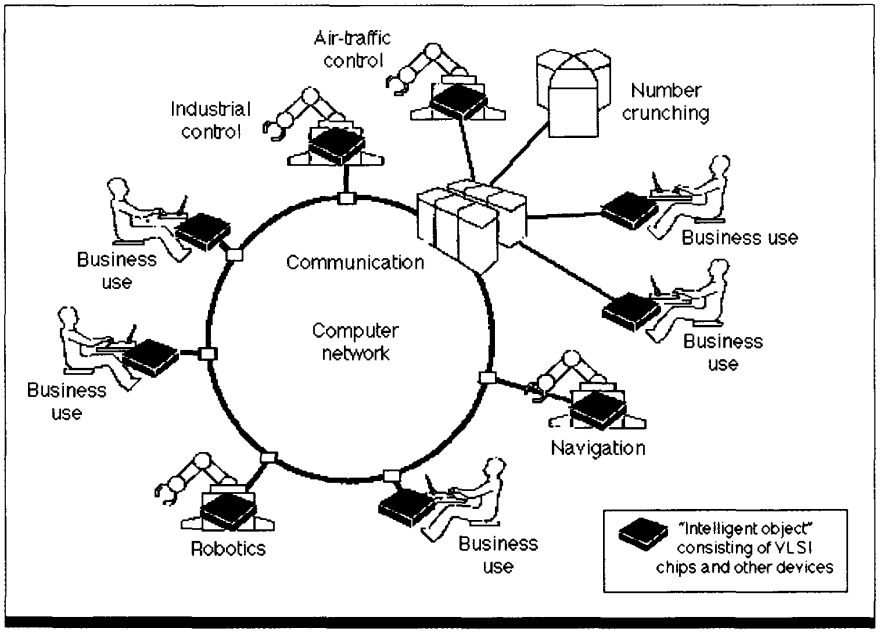


Figure 1. The standardized network structure linking "intelligent objects" advocated by the TRON project. Details of the construction can be found in Sakamura.[5] Standardized operating systems are proposed.

Floating-point hardware and software. Prior to 1960 computers used instructions to handle integers and programmers wrote floating-point software libraries utilizing these instructions. The programmers found it difficult to write a floating-point library that executed efficiently and yet was economical in memory usage. Mainframe manufacturers introduced special floating-point arithmetic instructions in the 1960s, drastically improving the execution speed and economy of memory. In the 1970s minicomputer vendors also started to supply floating-point instructions on high-end products. Unfortunately, each manufacturer choose different floating-point data formats. Exchanging the floating-point data and floating-point software between two different machines presented major difficulties.

To resolve the difficulties, in the late 1970s the Institute of Electrical and Electronics Engineers proposed a floating-point standard (now called the *ANSI/IEEE Standard 754-1985 for Binary Floating-Point Arithmetic*[6]). Microprocessor manufacturers have unanimously accepted this standard. It explicitly defines the essential hardware features of floating-point arithmetic, including the floating-point data format, semantics of operation, conditional branch mechanisms, and exception handling. All conforming systems give the same result for each operation.

Writing floating-point software in a high-level language, or HLL, increases the software's reliability, portability, and maintainability. However, when written in the current HLLs,[7] numerical software often suffers a decrease in reliability. It is impossible for a programmer to determine the exact floating-point operations that will be executed for a particular source-level construction of an HLL. The floating-point operations generated from an HLL source code remain unpredictable in the eyes of the programmers. Some of the mysteries often encountered in HLL numerical programming include the following:

Sign	Exponent	Mantissa

Figure 2. The components of a conventional floating-point number.

1) Commutative law ($x + y = y + x$) does not hold in many HLL compilers.

2) In Fortran one can test two variables for inequality at one point in a program and find that they are not equal, and then test them again later to find that they are equal.

3) A floating-point program often gives different results before and after a code optimization by an HLL compiler.

To eliminate the unpredictability and ambiguities of floating-point arithmetic in the C language, the American National Standards Institute released a draft proposal for the C language.[7] C when implemented according to this draft proposal makes the exact floating-point operations visible to the programmer.

Summary of Floating-Point Standards

One industry standard and one standard proposal define the boundary conditions of future floating-point hardware implementations. The purpose of both of them is the portability of floating-point software among various systems. One calls for more floating-point software to be written in a high-level language.

The *ANSI/IEEE 754-1985 Standard for Binary Floating-Point Arithmetic* explicitly defines essential functions of floating-point arithmetic hardware and software. However, it defines the functions as floating-point operations without regard to other software support such as that in high-level languages. The standard sets

- data formats (32-bit single, 64-bit double, and 80-bit extended-double precisions);
- six standard arithmetic operations (addition, subtraction, multiplication, division, square root, and remainder), requiring a precise solution for each;
- branches on floating-point conditions;
- binary and decimal conversion of floating-point data,
- rounding methods; and
- exception traps and handling methods.

The *Draft Proposed American National Standard for Information Systems—Programming Language C* determines the exact floating-point operations that will be executed for a particular source-level construction. The programmer can estimate what floating-point operations are generated just from the C source program. The draft proposal sets

- precisely defined rules for expression evaluation. In the type promotion rule the binary operators (+, −, *, /) must calculate the result in the data format of the operand(s) with the highest precision. Examples of the type promotion rule include:

<long double> + < double> –> <long double>
<float> * <double> –> <double>
<double> − <double> –> <double>

- strict rules to eliminate optimization resulting in the loss of exactness of operations,
- no change of operation precedence, and
- 21 well-defined mathematical functions.

ANSI is developing the draft proposal to "provide an unambiguous and machine-independent definition of the language C."[7] The standard lets a programmer determine the exact floating-point operations that will be executed by adjusting the source-level construction of C. Most of the mysterious phenomena seen in HLL floating-point programming can be eliminated by explicit rules for the precision of intermediate values in expression in the compiled code and rigorously defined, standard mathematical library functions. Most HLL compilers capriciously choose the precision for intermediate expression in the compiled code, which explains most of the mysterious phenomena described earlier. A programmer can now write numerical software without fearing the hidden "features" of an HLL.

Though the draft proposal sets no specific requirements on floating-point hardware, conventional floating-point units are not suited for efficient execution of a system based upon the draft proposal. Thus designers must give the floating-point hardware more dexterity in controlling the precision of the intermediate values to handle the exact operation demanded by the ANSI draft proposal. The Gmicro/FPU contains an architectural feature that adjusts to the draft proposal without impairing performance. See the box for a summary of the IEEE standard and the ANSI draft proposal documents.

The precision issue. The IEEE floating-point document standardizes the precision of basic mathematical operations such as add, subtract, multiply, divide, square root, et cetera. No precision requirement for the elementary functions is included; it is left up to the implementer. (Elementary functions are frequently used in scientific arithmetic and include trigonometric, hyperbolic, exponential, and logarithmic functions.)

When floating-point hardware is used in critical applications such as navigation, the precision of an elementary function matters a great deal. A fraction of error in a trigonometric function can lead you many miles away from your destination. Hardware designers cannot pass this problem to the software designers, since they are in a position to determine any trade-off between performance and precision. Since the software designers must compete with each other about the performance of the total system, they can hardly improve the precision at the cost of performance. On the other hand, the hardware designers are more likely to be the ones to choose the bit length of the arithmetic unit and the constants. They are also the ones to have the options in implementing special hardware without introducing deterioration of the performance.

Floating-point units

Most microprocessor manufacturers prefer to integrate the floating-point arithmetic functions on a sepa-

rate chip from the CPU known as a floating-point unit. Familiar FPU examples include Intel's i80387 designed for use with the i80386 CPU and Motorola's MC68882 designed to accompany the MC68020 CPU. There are two reasons for producing a CPU and FPU separately. The first is the system's requirement: Floating-point functions are not used in all systems. The second is the nature of VLSI production: The yield Y of a VLSI chip is expressed by

$$Y = a * e^{-D * S}$$

where D is the density of defects, S is the die area, and a is a coefficient factor common to all technology. The technology determines the density of defect D. For high-end, state-of-the-art microprocessors, the yield is very sensitive to small changes in the die area. Thus, implementing the entire CPU function and floating-point arithmetic calculations on one die would cause the yield of the chip to drop substantially, resulting in a more-costly chip.

Coprocessor interface. An instruction extension chip such as an FPU designed to accompany a VLSI CPU is often called a coprocessor. To reduce the number of interchip interconnections, designers build an autonomous control system into the extension chip, making it a cooperating processor rather than an extension of the CPU's logic. The instructions extended by the coprocessors are called coprocessor instructions. When a VLSI CPU encounters a coprocessor instruction, it generates a series of handshake procedures on the coprocessor chip. These procedures, called coprocessor protocols, perform the transfer of the instruction information, data, and state, while leaving the execution of the instruction to the coprocessor. The CPU normally does not "know" the content of coprocessor instruction operation, leaving the detailed definition of coprocessor instructions to a later time when the coprocessor is actually implemented.

The TRON architecture for the VLSI CPU includes 23 coprocessor instructions, which cover all probable interactions with coprocessors to be defined in the future. Since the coprocessor protocol is not part of the architecture and its implementation is left to the manufacturers, the protocol can be defined so that it is optimum to the specific technology in which the chip is fabricated.

The Gmicro VLSI CPUs such as the Gmicro/200 and Gmicro/300 include coprocessor instructions. Programmers see the FPU instructions the same way they see other CPU instructions. By connecting the Gmicro FPU, we add 50 floating-point-related instructions and 19 floating-point-related registers to the Gmicro instruction set, the register resource. Vendors producing the Gmicro family of VLSI chips have defined a coprocessor protocol common to the Gmicro family.

The coprocessor protocol between the CPU and FPU determines much of an FPU's performance. The speed of bare arithmetic hardware, such as the arithmetic logic unit or the multiplier, does not become the performance determinant so long as the chip is not equipped with a faster coprocessor protocol. Manufacturers have implemented various kinds of handshake methods between the CPU and the extension. Some coprocessors scan the instruction stream, identify their own instructions, and activate themselves. Other coprocessors are mapped onto a special memory space so the CPUs can access them as slave processors. The protocol of the Gmicro/FPU and Gmicro VLSI CPUs are the latter type except that some special interface pins are added to decrease the number of bus cycles necessary to complete the coprocessor protocol. The number of bus cycles is a major determinant of the maximum performance of the FPU, since the arithmetic operations can now be performed in very few machine cycles.

New computation algorithms. With advances in technology, vocabularies of the VLSI architecture gradually evolve, and the cost of each numerical operation changes. As a result designers discard traditional algorithms in favor of more appropriate ones such as the Cordic (COordinate Rotation Digital Computer) algorithm for calculating a wide range of elementary functions. Cordic's implementation with a set of adders, shifters, and read-only memory allows most of today's VLSI FPUs to use the silicon area to best advantage.

Instances of an algorithm inspired by a new hardware technology can be found in the Gmicro/FPU. Designers introduced a parallel multiplier consisting of a matrix of carry-save-adders, which multiplies two operands in two or three machine cycles. Previously, when implemented in a microinstruction loop, multiplication took 30 to 70 machine cycles. The drastic reduction in the computation cost of multiplications makes room for new algorithms for IEEE square-root and elementary functions. The new algorithm is potentially faster than any known algorithm when fast multiplication is available.

Requirements

The design objectives for the Gmicro/FPU floating-point unit were set to:

- fully conform to the IEEE floating-point standard,
- provide adequate architectural support for the ANSI C draft proposal,
- provide an efficient CPU interface as well as high-performance floating-point hardware,
- provide elementary functions with the highest precision, and
- provide an entire set of mathematical library functions as instructions.

Architecture

The Gmicro/FPU can be used as a resource of the Gmicro/200. Figure 3 shows the extended register after the FPU is attached to the Gmicro/200. The FPU contains sixteen 80-bit, floating-point data registers (FR0-FR15) and three control registers: the FMCR (floating-point mode), FSR (floating-point status), and FQR (floating-point quotient). The registers function as follows:

- FR0-FR15 registers store floating-point data;
- the FMCR specifies trap enable/disable and rounding modes;
- the FSR stores condition codes, exception flags, and accrued exception flags; and
- the FQR stores the quotient for modulo and remainder instructions.

Instruction set. The FPU's instruction set and its basic formats appear in Table 1 and Figure 4. All instructions for arithmetic operations have only floating-point registers for destinations. Figure 4 displays the major instruction formats. Designers modeled the basic instruction format after the coprocessor instructions in the TRON architecture for the VLSI CPU. Both source and destination operands have size specifiers, which give the unit an unusual architectural strength. The result is rounded to the precision of the destination operand size, exception signaled (if any), and stored in the floating-point data register.

Figure 5 illustrates how, when variables are assigned in floating-point registers, an ANSI C compiler can generate faster code for the Gmicro/FPU than can be obtained with conventional FPUs. The C-language source code in Figure 5a on p. 32 is cited as an example of expression evaluation. According to the ANSI proposal, the evaluation of the expression $u = t * (x + y) + z$ takes the four steps shown in Figure 5b; in each step, the rounding precision is explicitly specified. Both source and destination operands have size specifiers, which enables the unit to conform to the ANSI C draft proposal even when the values reside in its floating-point registers. The earlier box contains the type of promotion rule that determines this precision.

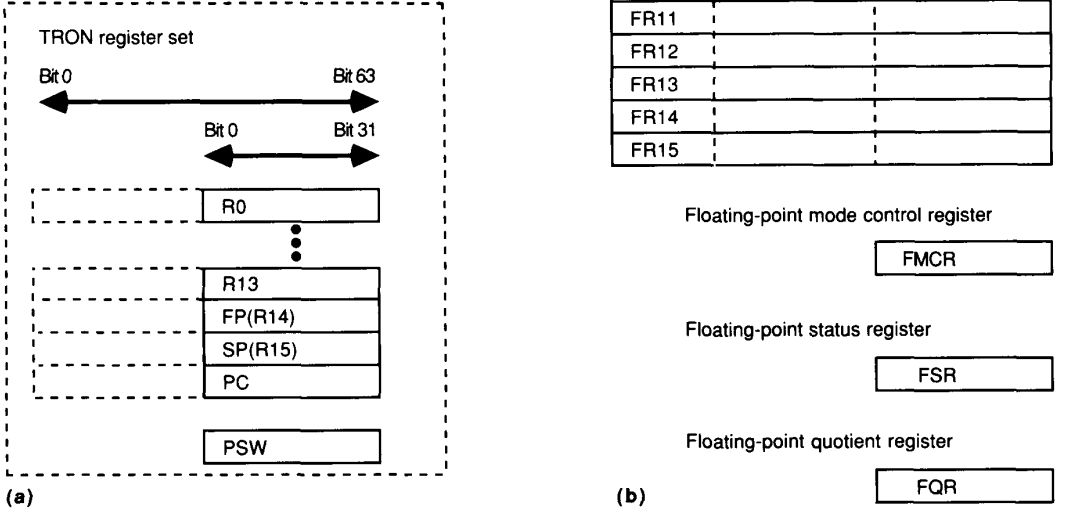


Figure 3. The TRON register set (a) and the extension of the register resource when the FPU is added to the Gmicro chips (b).

A common tendency in the current floating-point chips is the precision change that occurs when floating-point registers are used in optimization.[4] Most of these chips convert results to the length of the registers—the longest data type a register can hold. Promotion of the floating (32-bit) variable to a long double (80-bit) variable occurs when the destination is a floating-point register. To observe the ANSI proposal's requirement on conventional floating-point chips, we would have to generate the object code shown in Figure 5c, a succession of 10 instructions. The .s and .d suffixes for the assembler mnemonic in this figure indicate single precision and double precision.

In Figure 5d we see the ANSI proposal for object code generation observed on the Gmicro/FPU. Here, only five instructions evaluate the expression, thereby reducing by 50 percent the amount of code necessary with a conventional FPU.

Elementary function instructions. The ANSI C proposal also determines the range of mathematical functions. Mathematical functions defined in the header file <math.h> take double-precision arguments and return double-precision values. The tenet of the function definition is that the domain of the mathematical function must match with the mathematical definition. The Gmicro/FPU instructions conform to this proposal. No software envelope is necessary, and an instruction for an elementary function can simply be generated in line when the functions are called. The complete matching of the domain of the function is ensured. Table 2 on p. 33 lists the correspondence of the instructions and the <math.h> file. Future ANSI expansions are expected to include all operand precisions in elementary functions. This requirement can be met by having the FPU specify the source and destination size in the instruction.

The table also shows the mathematical library function of Fortran 77, which has complex numbers as a data type. Mathematical functions dealing with a complex data type can easily be constructed by using the Gmicro/FPU elementary function instructions. However, this is out of our scope and not shown in the table.

Table 1.
Gmicro/FPU instruction set.

Operation	Function
Basic	Addition, subtraction, multiplication, division, remainder, IEEE modulo, square root
General	Absolute value, negation, round to integer, truncate to integer, extract exponent, extract mantissa, generate constant
Elementary functions	Sine, cosine, sine cosine simultaneous calculation, tangent, arcsine, arccosine, arctangent, sine hyperbolic, cosine hyperbolic, tangent hyperbolic, arctangent hyperbolic, exponent base 2, exponent base 10, natural exponent, logarithm base 2, logarithm base 10, natural logarithm
Graphics support	Vector inner product (up to eight dimensions), clipping judgment
Conditional branches	Compare, test, conditional branch
Data format conversion	Load floating-point number, store floating-point number, load signed integer, store signed integer, load unsigned integer, store unsigned integer, load control register, store control register
Operating system related	No operation, internal reset, save internal state, restore internal state, load multiple floating-point numbers, store multiple floating-point numbers

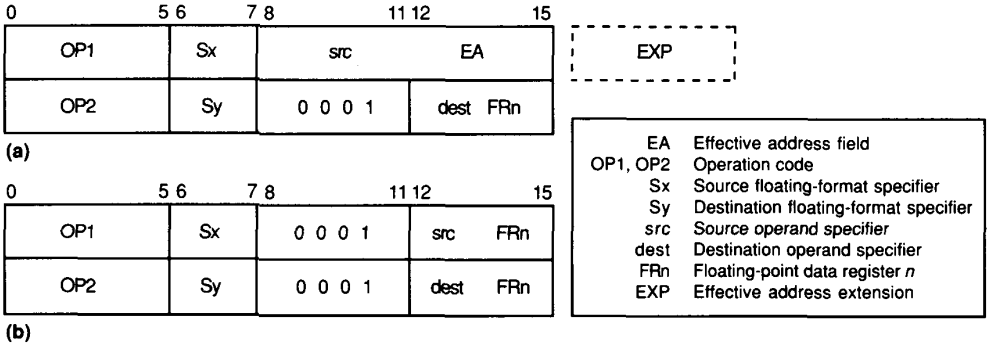


Figure 4. Major instruction formats in the Gmicro/FPU: memory-register (a) and register-register (b). All arithmetic instructions fall under these categories, and only store instructions can store to memory.

```
main()
{
        register float          x, y, z, u;
        register double         t;
        {
                ...
                u = t • (x+y)+z;
        }
}
```

(a)

1) $x + y$ is performed to infinite precision and rounded to float or single (32-bit) precision.

2) t • [result of (1)] is performed to infinite precision and rounded to double (64-bit) precision.

3) [result of (2)] + z is performed to infinite precision and rounded to double (64-bit) precision.

4) [result of (3)] is rounded to single precision and stored in u.

(b)

```
sp: stack pointer
fr15 : x
fr14 : z
fr13 : y
fr12 : t
fr11 : u

fmov.s          fr15, fr0       ; move x to fr0
fadd.s          fr13, fr0       ; add y and store in fr0
fmov.s          fr0,@sp         ; store result to round to single
fmov.s          @sp, fr0        ; load expression x+y in fr0
fmul.d          fr12 , fr0      ; multiply t with content of fr0
fmov.d          fr0, @sp        ; round the result to double
fmov.d          @sp, fr0        ; load expression t • (x+y) in fr0
fadd.s          fr14, fr0       ; add z  to fr0
fmov.s          fr0, @sp        ; round the result to single
fmov.s          @sp, fr11       ; store the result to u

10 instructions
```

(c)

```
sp : stack pointer
fr15 : x
fr14 : z
fr13 : y
fr12 : t
fr11 : u

fmov        fr15, fr0.s         ; move x to fr0 (accumulator)
fadd        fr13, fr0.s         ; add y to fr0, round to float
fmul        fr12, fr0.d         ; multiply t, round to double
fadd        fr14, fr0.d         ; add z, round to double
fmov        fr0, fr11.s         ; round fr0 store to u

5 instructions
```

(d)

```
.s  Single precision
.d  Double precision
```

Implementation

Here we describe a new, tightly coupled coprocessor protocol for the Gmicro/200-Gmicro/FPU system. One of the design objectives in this protocol is to minimize the CPU-FPU communication overhead to improve the total performance of the FPU. To achieve this goal, we judiciously reduced the amount of CPU, FPU, and memory bus transfers needed for instruction execution. We introduced the following schemes in the protocol:

1) *Some functional redundancies between the CPU and the FPU.* The decoder and the microcode of both the Gmicro/CPU and the Gmicro/FPU store the coprocessor instruction and protocol sequence. Both chips "know" exactly what is to be done once the coprocessor instruction is determined. This capability significantly reduces the bus cycle count over that found in other FPUs, since no coprocessor instruction knowledge must be transferred from the coprocessor to the CPU.

2) *Protocol reduction.* Coprocessor status pins (CPST) reduce the number of bus cycles in a protocol. The Gmicro/CPU can monitor the Gmicro/FPU's status without having to invoke a bus cycle.

3) *Direct data transfer.* The Gmicro/200's bus control circuitry directly transfers data between the memory and the Gmicro coprocessor. Eliminating data transfers between the CPU and the coprocessor normally required for memory-coprocessor data transfer again reduces the number of bus cycles required for the protocol.

4) *CPU write-through cache support.* The Gmicro/FPU supports a write-through cache on future CPUs. Securing the data coherency between the Gmicro/CPU's internal cache and external memory becomes an easy task.

5) *Multiple coprocessor support.* Up to eight coprocessors can logically connect to the CPU. In this way, the CPU can distribute the work load to several coprocessors, making overlapping operations on multiple coprocessors possible.

6) *Fast floating-point conditional branch.* Conditional branch tests on the coprocessor's internal state take place using the coprocessor status lines, thereby making the operation faster.

An example multiple-coprocessor system. Figure 6 depicts a multiple-coprocessor system made up of Gmicro family chips. It contains four coprocessors including an FPU. A common clock serves both the CPU and the coprocessors as the time reference for

Figure 5. Two operand size specifiers on a floating-point instruction conform to the ANSI C draft proposal very simply when the compiler allocates variables in the floating-point registers. C language source code (a); its translation into a floating-point number according to the ANSI standard (b); object code on conventional FPUs (c); and the Gmicro/FPU optimization (d).

Table. 2.
Mathematical libraries and instructions for the Gmicro/FPU.

Instruction	ANSI/C proposal math.h	Fortran 77/ANSI	Instruction	ANSI/C proposal math.h	Fortran 77/ANSI
fabs	fabs	ABS	fsincos,fdiv	—	COTAN
facos	acos	ACOS	fsinh	sinh	SINH
fasin	asin	ASIN	fsqrt	sqrt	SQRT
fatan	atan	ATAN	ftan	tan	TAN
fatan,fdiv	atan2	ATAN2	ftanh	tanh	TANH
fcos	cos	COS	floge	log	LOG
fcosh	cosh	COSH	flog10	log10	LOG10
fexp	exp	EXP	fmod	modf	MOD
fexp2	frexp	—	—	pow	—
fint,fintrz	floor, cell	AINT,ANINT, NINT	fmul	*	*, DPROD
			fneg	-	SIGN
fscale	ldexp	—	fsqrt,fsincos, fatan	—	SQRT
fsin	sin	SIN			

synchronized operations. The synchronized clock enables all of the processors to operate essentially as one processor. Each coprocessor contains input signals called coprocessor identifications, or CPIDs, to uniquely assign an identification for user software. The system initializes the CPID in the reset operation. All processors share the data bus, address bus, and part of the control bus. Notable control signals include bus accesses BAT0-2 and coprocessor status CPST0-2. The CPST0-2 and the BAT0-2 for each coprocessor are OR-wired electrically or logically.

Bus access signals BAT0-2. The Gmicro/CPU uses the bus access signals to inform the Gmicro/FPU of contents on the data bus. The CPU

1) sends an operation's content to the coprocessor,
2) writes the operand data into the coprocessor,
3) reads the operand data from the coprocessor,
4) coordinates the direct data transfer from the coprocessor to the memory,
5) coordinates the direct data transfer from the memory to the coprocessor, and

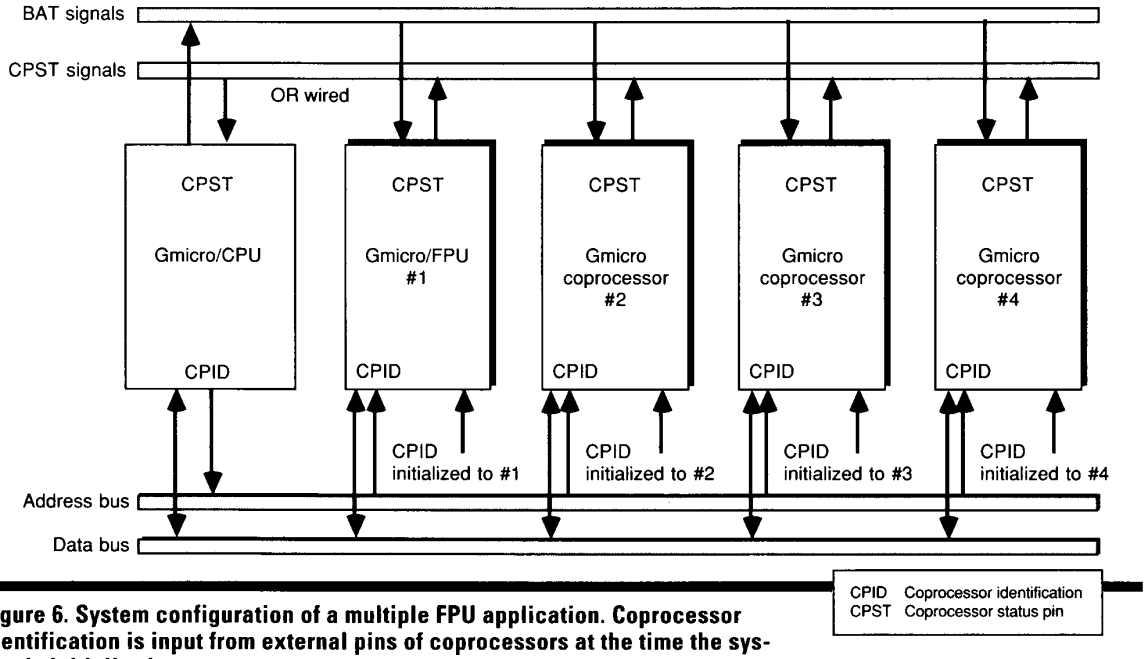


Figure 6. System configuration of a multiple FPU application. Coprocessor identification is input from external pins of coprocessors at the time the system is initialized.

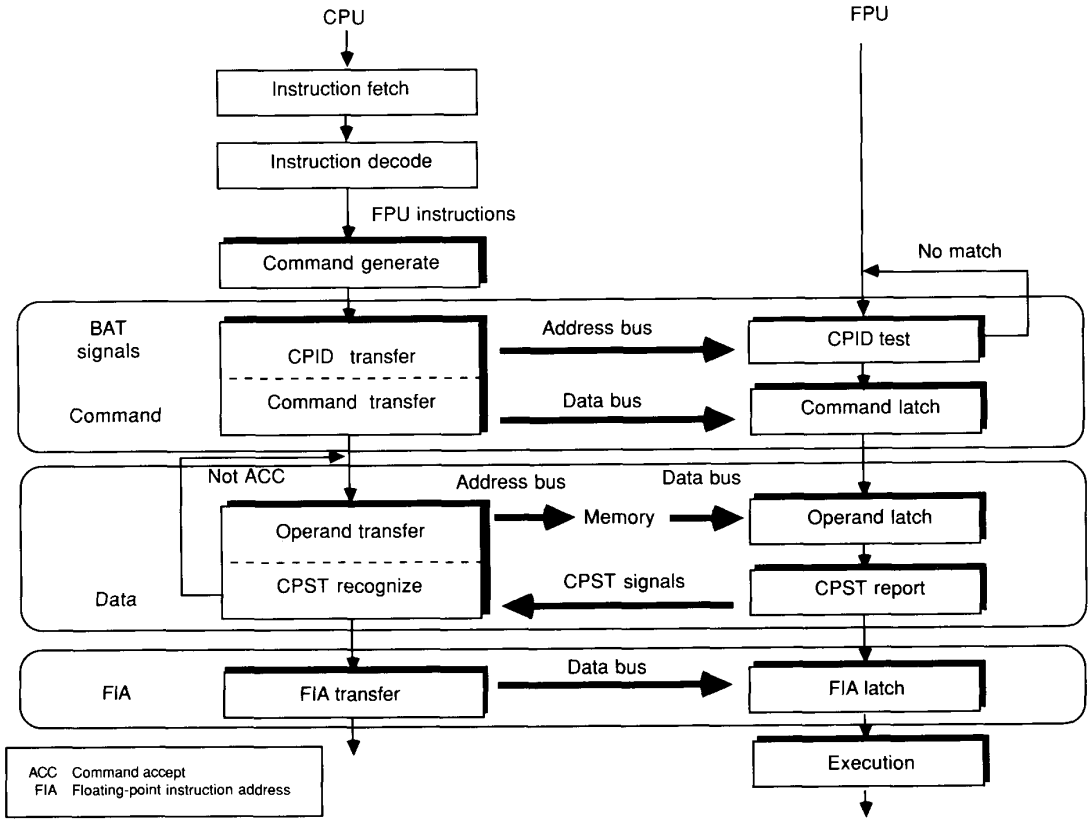


Figure 7. Protocol sequence in the Gmicro/FPU when executing a coprocessor instruction that loads a single-format operand and completes an arithmetic operation.

6) sends the coprocessor instruction address to the coprocessor.

The BAT0-2 signals along with a read/write signal from the CPU determine which one of the above bus access types are currently taking place. From this information, the Gmicro/FPU sends, receives, or processes data on the data bus for each bus access type.

Coprocessor status signals. The CPST0-2 signals report the coprocessor's status to the Gmicro/CPU. The sampling timing of the CPST0-2 signals is determined within the protocol. Normally, these signals are sampled on the second bus cycle in the protocol. The first bus cycle is the operation content transfer. In this case the coprocessor compares its internal state and the operation content to see if the operation can be processed immediately and returns the CPST0-2 signals. The status that a coprocessor could present over the CPST0-2 signals include:

1) *Command acceptance.* The operation required by the CPU is acceptable and immediately processed.

2) *Command error.* The coprocessor does not recognize the operation.

3) *Coprocessor busy.* The coprocessor cannot accept the operation, and therefore the CPU must redo the first two bus cycles in the coprocessor protocol.

4) *Coprocessor exception.* The coprocessor detects an exception in the previous operations, and therefore the CPU must take an exception vector.

5) *Data transfer ready.* The coprocessor is ready to accept or send data operands. This status is reported for the execution of a coprocessor instruction involving multiple operands.

6) *Condition true.* The conditional branch test on the coprocessor state is true.

7) *Condition false.* The conditional branch test on the coprocessor state is false.

Basic types of protocols. A combination of the data, command, and address transfers on the bus cycles and status tests in a certain order form the protocol sequence for a coprocessor instruction. Twelve kinds of protocols perform Gmicro/FPU instructions including internal state save/restore, multiple-register load/store,

and floating-point arithmetic. The floating-point arithmetic operations fall into three groups:

- an operation on the FPU's internal registers only;
- an operation using an operand(s) external to the FPU, CPU register, memory, or CPU cache and storing the result in the FPU's internal register(s); and
- an operation using an operand(s) internal to the FPU and storing the result in the resource external to the FPU, memory, or CPU register.

A coprocessor protocol example. Figure 7 illustrates a coprocessor protocol for a Gmicro/FPU dyadic instruction, an instruction involving two operands to produce the result in an operand. One operand resides in the memory and the other in the Gmicro/FPU's register. This example helps to explain the coprocessor protocol. In executing this coprocessor instruction, the Gmicro/CPU and Gmicro/FPU perform the following activities:

- *Instruction decoded, operation content extracted.* The CPU fetches an instruction and decodes it. If it is a floating-point instruction, the CPU extracts the information needed by the FPU from the coprocessor instruction.
- *Operation content transferred to the FPU along with the coprocessor ID.* The FPU transfers the operation content and command. When this bus cycle completes, three of the address lines indicate the value of the CPID number to which this coprocessor instruction is sent. The coprocessors monitor the three address lines and compare the CPID number on the address lines with their ID numbers. The coprocessor whose CPID matches with the number acknowledges the command and asserts the CPST0-2 lines. Other coprocessors designate their coprocessor status lines as three-state conditions.
- *The CPU coordinates the operand transfer from the memory to the FPU.* The CPU asserts the bus control lines and the address lines and instructs the main memory to put an operand word on the data bus. At this point the FPU asserts the CPST0-2 lines, and the CPU samples them. The FPU receives the operand on a data bus, and the CPU proceeds with the protocol if it detects the command acceptance status. If a coprocessor busy status is detected, the CPU reverts to activity 2. If another coprocessor status is presented, exception actions are taken. CPST0-2 detection and the operand transfer occur simultaneously. If the operand transfer requires more than one bus cycle, either because the size of the operand is larger than 32 bits or because the operand is placed out of alignment with the memory, the bus cycles repeat until the operand transfer completes.
- *The CPU transfers the FPU instruction address to the FPU.* The CPU transfers the floating-point instruction address via the data bus from which the FPU receives the FPU instruction address. This step ensures the availability of the exception information necessary in case an error occurs at a later stage. When the FPU acknowledges the instruction address transfer, the protocol sequence terminates.

Bus-cycle reduction. Figure 8 replicates the actual time chart used for the protocol when the FPU performs a floating-point arithmetic operation of the kind just mentioned: one operand of single-precision data format residing in memory and one operand in floating-

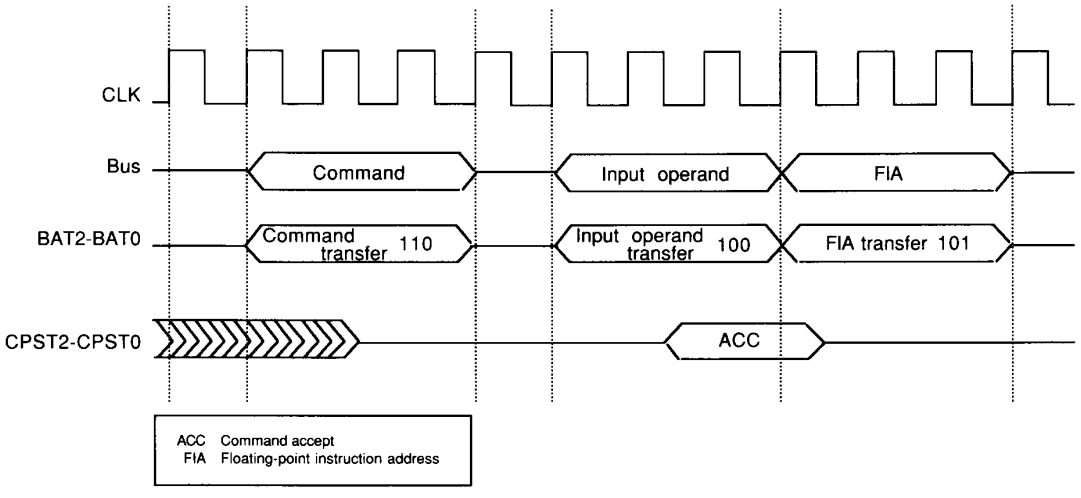


Figure 8. FADD command-cycle timing (memory + FR). A clock cycle equals the machine cycle. The Gmicro/220-Gmicro/FPU system executes this protocol in 10 machine cycles.

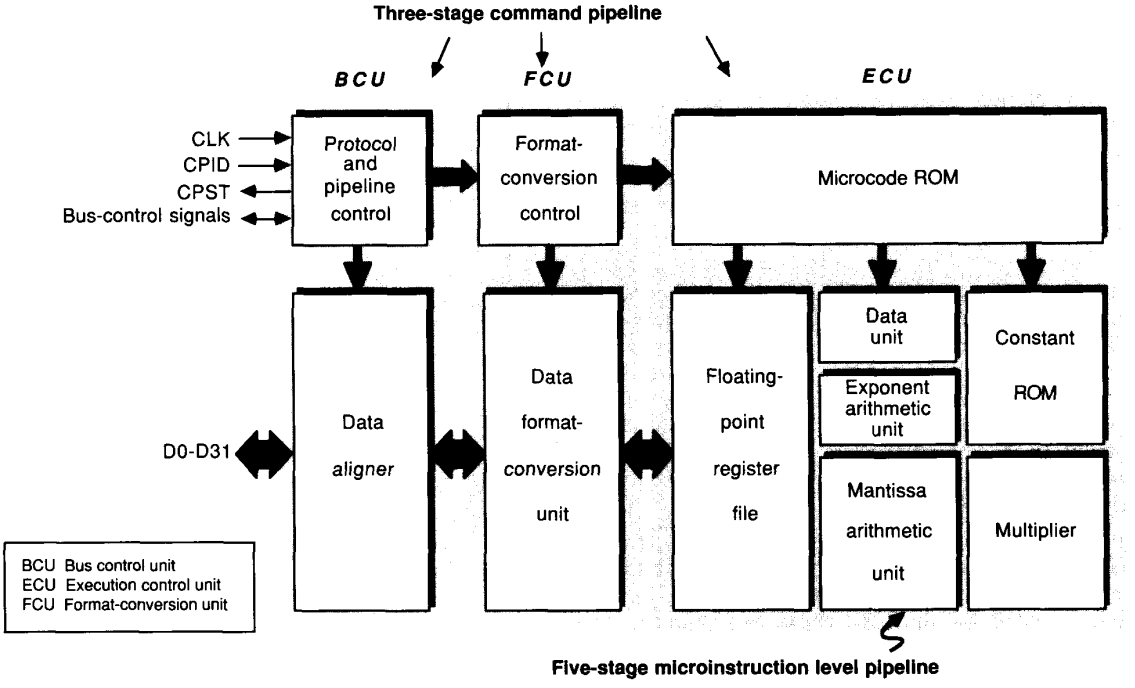


Figure 9. Internal structure of the Gmicro/FPU. The BCU, FCU, and ECU units constitute the active operational blocks, and another piece of logic arbitrates them.

point register with the result stored in a floating-point register. This operation takes 10 machine cycles to complete the handshake.

In this specific protocol the coprocessor status line saves one bus cycle, and direct data transfer between the memory and the Gmicro/FPU saves another bus cycle. Thus we save a total of two bus cycles from the original five—a 40 percent reduction. This amount of reduction is entirely effective since 10 machine cycles are also consumed by the floating-point addition for a single-precision operand. The new protocol scheme matches bandwidths between the high-performance floating-point arithmetic hardware and the protocol.

Other aspects. The FPU can also be used with a CPU without Gmicro coprocessor protocol when it is connected as a peripheral. The CPU emulates the transfers in the protocol sequence of coprocessor operation with the exception that the coprocessor status is read from a special register address. For future Gmicro/CPUs with nonwrite-through data caches, the Gmicro/FPU can retain the data coherencies between the external memory and the CPU cache. When the correct data is in the memory, the CPU can activate the bus cycle from the memory to the FPU. When the correct data is in the CPU's data cache, the CPU can write the data into the FPU. The FPU does not differentiate between the data access methods, carrying out the protocol in the same way wherever the operand originates.

Internal FPU structure. Figure 9 shows the Gmicro/FPU's internal structure. The three main elements are the:

- *Bus control unit (BCU)*, which is responsible for handshaking with the main processor. The unit carries out bus cycles, generates the status of the coprocessor, and processes the coprocessor protocol. It also carries out the necessary command/data/status transfers between the main processor and the FPU.
- *Format conversion unit (FCU)*, which converts all operands sent from the main processor or the memory to the internal floating-point data format before the actual arithmetic operation starts. In the FPU one internal data format represents all floating-point data. Once the store operation of the floating-point data completes, the data expressed in the internal format are again converted into the external formats, as defined in the IEEE standard.
- *Floating-point execution unit (ECU)*, which consists of the microcode ROM and four other units: data type, exponent arithmetic, mantissa arithmetic, and multiplier. The IEEE floating-point data format expresses those values such as infinity, zero, and not-a-

number. The arithmetic operations involving these special values bypass normal sequencing and occur in the data unit. The sign bit also resides in this unit and is manipulated here. The exponent arithmetic unit processes addition/subtraction operations for the exponent calculations. The addition, subtraction, shifting, division, and other necessary operations are performed on the mantissa of floating-point values in the mantissa arithmetic unit. The quarter-size flash multiplier (33 bits × 33 bits) performs high-speed multiplication operations on the mantissa of the floating-point numbers. This multiplier unit is used frequently when calculating square roots, elementary functions, vector inner products, as well as multiplications operations.

Three-stage pipeline. In addition to the coprocessor interface with the reduced bus cycle overhead, the Gmicro/FPU controls a three-stage pipeline. Often the entire protocol processing becomes hidden in a floating-point arithmetic operation. Figure 10 illustrates how the three-stage pipeline control increases the throughput of the FPU by allowing command or data fetches in the BCU, data conversions in the FCU, and floating-point arithmetic operations in the ECU to be performed concurrently.

Excluding the instruction fetch and effective address calculation by the CPU, a floating-point instruction execution in the FPU can be divided into the following three phases:

- the BCU fetches the command and operand, the BCU fetches the FIA (floating-point instruction address);
- the FCU receives the operand from the BCU and converts IEEE format data into internal data format; and
- the ECU performs floating-point arithmetic.

By overlapping the operations for different floating-point instructions, each unit operates in parallel. The pipeline arbitration occurs in a finite-state machine that samples the state of each unit. The status of each instruction at any one time (see Figure 10) in the pipeline is as follows:

- the first instruction is in the ECU processing phase;
- the second instruction is in the FCU processing phase; and
- the third instruction is in the BCU processing phase.

Elementary functions. Designers know that their goals of high performance and high precision are often at odds with each other when designing floating-point units. Repeating iterations of converging algorithms ensures higher precision, yet the use of the same computational algorithm takes longer to execute. In pursuit of our design objective of higher precision in elementary functions, we attempted to streamline the Cordic algorithm to avoid lowering performance for precision. In realizing the elementary functions, we

1) adopted the Cordic algorithm because it lets us realize the entire range of elementary functions with a simple set of hardware.
2) added the adjustment mechanism to, for each exponent of the argument, always select the best set of angle constants used for rotations from a constant pool.
3) corrected errors after applying the Cordic algorithm to obtain higher precision with decreased Cordic iteration count.

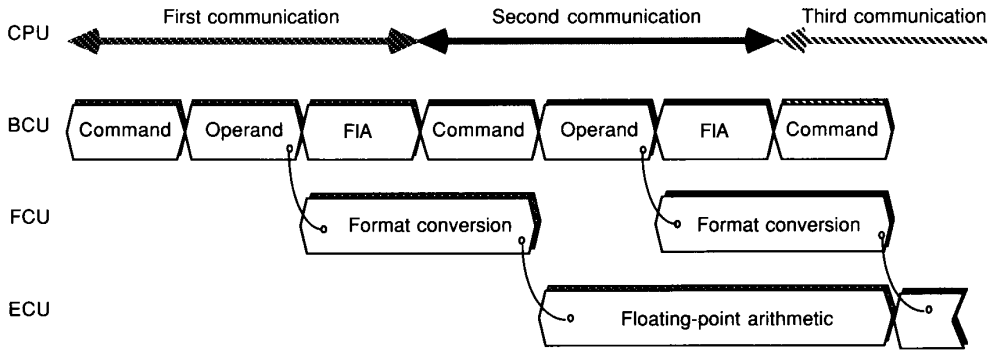


Figure 10. Pipeline timing in the Gmicro/FPU showing the CPU as it executes three consecutive memory-to-register floating-point arithmetic instructions. The length of each stage differs from one another. The format-conversion stage can start from the middle of the protocol. A state machine, which schedules each pipe to near-maximum efficiency, manages the pipeline.

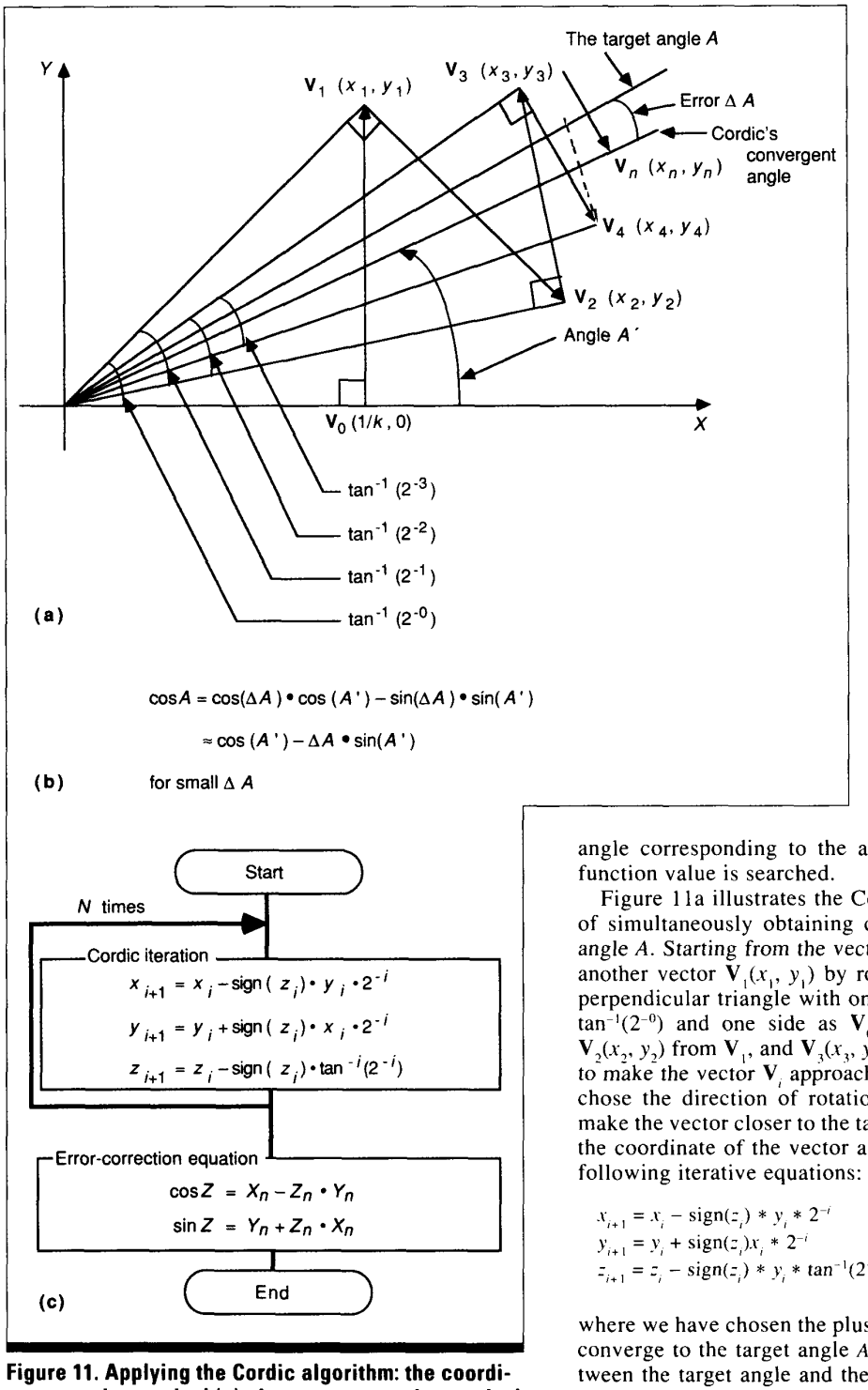


Figure 11. Applying the Cordic algorithm: the coordinate-rotation method (a); the error-correction method for a trigonometric function (b); and convergence improvement after replacing the last half of the Cordic iterations with a faster method (c).

In the following paragraphs we use the cosine function as an example to demonstrate how we implemented the Cordic algorithm.

Original Cordic algorithm. The Cordic algorithm invented by Chen[8] was expanded by Volder[9] and Walther[10] to be applied for the entire range of elementary functions. The algorithm's very wide range of functions even permits a multiplication to be performed with Cordic hardware. In the Gmicro/FPU, however, we use a separate multiplier for multiplication. The Cordic method obtains the solution by repeating vector rotations in the complex plane and eventually lets the vector converge to the angle corresponding to the argument for which the function value is searched.

Figure 11a illustrates the Cordic algorithm process of simultaneously obtaining cosine and sine for the angle A. Starting from the vector $\mathbf{V}_0(1/k, 0)$, we create another vector $\mathbf{V}_1(x_1, y_1)$ by rotation; we then form a perpendicular triangle with one vertex with the angle $\tan^{-1}(2^{-0})$ and one side as $\mathbf{V}_0$. Similarly, we obtain $\mathbf{V}_2(x_2, y_2)$ from $\mathbf{V}_1$, and $\mathbf{V}_3(x_3, y_3)$ from $\mathbf{V}_2$, and so forth to make the vector $\mathbf{V}_i$ approach the target angle A. We chose the direction of rotations in each iteration to make the vector closer to the target angle A. We obtain the coordinate of the vector after this rotation by the following iterative equations:

$$x_{i+1} = x_i - \text{sign}(z_i) * y_i * 2^{-i} \quad (1)$$

$$y_{i+1} = y_i + \text{sign}(z_i) x_i * 2^{-i} \quad (2)$$

$$z_{i+1} = z_i - \text{sign}(z_i) * y_i * \tan^{-1}(2^{-i}) \quad (3)$$

where we have chosen the plus and minus operators to converge to the target angle A. ΔA, the difference between the target angle and the vector $\mathbf{V}_n$, is translated into the error in the final solution. The maximum value of ΔA, $\max(|\Delta A|)$, becomes smaller by one bit with each rotation and eventually becomes less than 2^{-n} after n iterations:

$$\max(|\Delta A|) \cong 2^{-n} \quad (4)$$

To converge the rotational vector $\mathbf{V}_n$ to the target angle A with 64-bit precision, the same as that of extended double-precision format, the algorithm requires 64 iterations.

Cordic becomes faster. As the above argument shows, in the Cordic algorithm, we obtain only one significant bit of rotational angle per rotation. To obtain the solution more quickly, we leave the Cordic iterations after a certain number of rotations and obtain the cosine for the angle of a rotational vector $\mathbf{V}_n$, A'. This vector occurs in the vicinity of the target angle A with the difference ΔA. ΔA being sufficiently small, the following identity holds:

$$\begin{aligned}\cos(A) &= \cos(\Delta A) * \cos(A') - \sin(\Delta A) * \sin(A') \\ &\cong \cos(A') - \Delta A * \sin(A') \\ &\text{for small } \Delta A\end{aligned} \quad (5)$$

Using the identity in Equation 5, we obtain $\cos(A)$. The Cordic algorithm gives $\cos(A')$ and $\sin(A')$. We get ΔA from simple subtraction. Thus we can effectively replace the remainder of the Cordic iterations by a subtraction and a multiplication, which take very little time to execute with the Gmicro/FPU's multiplier.

Implementation, performance, precision. The Gmicro/FPU repeats the Cordic iterations 32 times and performs the error-correction operations seen in Figure 11b to obtain the cosine function. For all other functions including sine, tangent, and hyperbolic functions, similar error-correction equations exist, and with these the Cordic iterations repeat only 32 times.

Figure 11c summarizes the comparison of the vital statistics of the original Cordic algorithm and the improved Cordic algorithm by examining the cosine function with extended double-precision (80-bit) format. The Gmicro/FPU can execute one Cordic iteration in three machine cycles. By reducing the Cordic iteration count from its typical 64 times to 32 times, we save 96 machine cycles. The error-correction operation consisting of subtraction and 64 × 64 multiplication (which takes one machine cycle and 12 cycles respectively) can be overlapped with other operations. Thus the total microcode of the new algorithm takes 111 machine cycles as opposed to the conventional Cordic algorithm that takes 198 machine cycles. We achieve a performance improvement of 40 percent. See Table 3.

The preliminary evaluation of the precision tells us that the cosine function has, in most of its domain, 58 to 62 significant digits out of the 64 mantissa digits of extended double-precision data. Most libraries or floating-point processors have 53 to 58 significant digits. The difference is accounted for as follows:

- the Gmicro/FPU has 66-bit-long angular constants for Cordic operation, whereas most software libraries typically have 64-digit angular constants; and

Table 3.
Cordic algorithm speed improvement on the cosine function.

Items	Conventional Cordic	Gmicro/FPU Cordic
Cordic iteration count	64	32
No. of multiplications	0	1
No. of significant digits	53-58 (out of 64)	58-62 (out of 64)
Latency times (machine cycles)	198	111

- the rounding-off error tends to accumulate with repetitive additions toward the end of the Cordic iterations. Thus, from a precision standpoint, the new algorithm demonstrates a satisfactory result.

Square roots. Square-root derivation is another area in which we can exploit the existence of the multiplier. With the Newton-Raphson algorithm we reach square roots extremely fast when fast multiplication is available. However, the IEEE floating-point standard sets a stringent requirement on the precision of the division and square-root solutions, which the N-R algorithm does not fulfill. A so-called precise solution is required for these operations. The IEEE standard states that the add, subtract, multiply, divide, square-root, and remainder operations "shall be performed as if it first produced an intermediate result correct to infinite precision and with unbounded range, and then rounded that result . . . ."[11]

It has been believed that the above requirement essentially determines the computation algorithm for square-root and division operations. Most manufacturers today use the pencil-and-paper algorithm for these operations, which essentially obtains one significant bit of solution per iteration and which provides a set of intermediate results from which an ALU with infinite precision is effectively simulated.

Instead, the Gmicro/FPU uses the new algorithm to provide an IEEE square root with a smaller number of iterations. The FPU's N-R algorithm obtains a square-root solution in a certain neighborhood of the IEEE square root, which we call a *pseudo square root*. (See the following explanation.) It then reconstructs the intermediate result of the pencil-and-paper algorithm from the pseudo square root and begins the pencil-and-paper algorithm for the last several bits to obtain the IEEE square root.

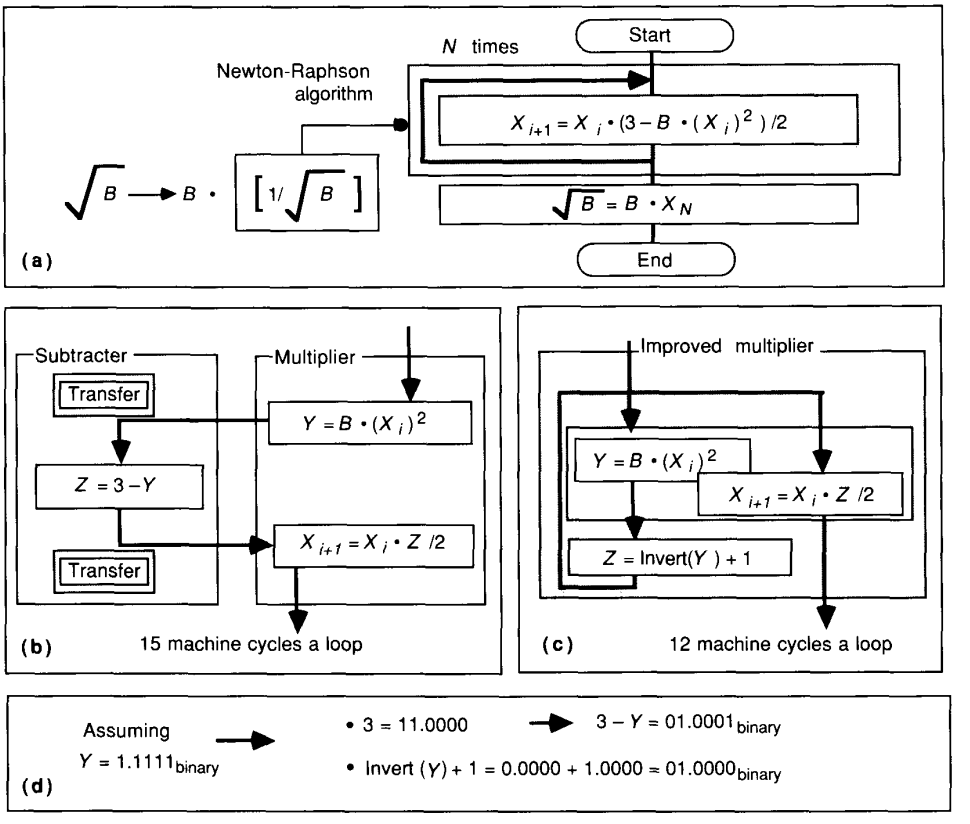


Figure 12. Square-root calculation. This technique for simplifying the Newton-Raphson method (a) saves three machine cycles in each iteration. The faithful evaluation in (b) is reduced further in (c). An example of 3 – *Y* = Invert(*Y*) + 1 (d).

Figure 12a shows the method of square-root extraction using the N-R principle. The N-R algorithm can obtain the reciprocal of the square root from an appropriate value by repeating Equation 6.

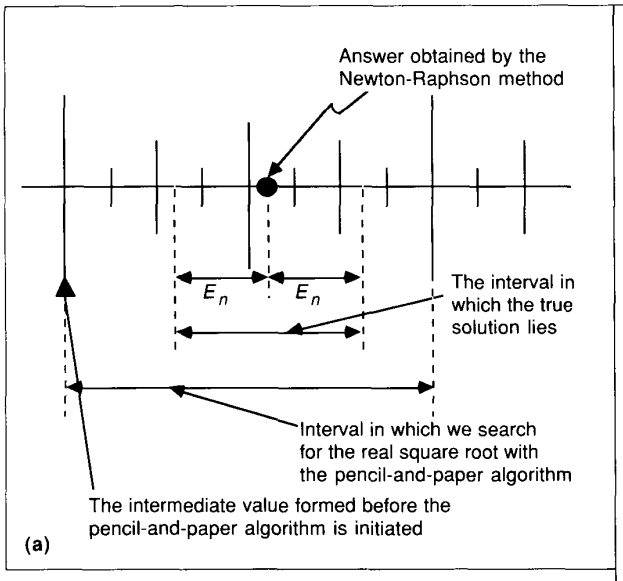
$$X_{i+1} = X_i * [3 - B * (X_i)^2]/2 \quad (6)$$

where B is the source operand for the square root. To make the convergence faster, we look up in the ROM table an approximate value with 8-bit precision. With only three N-R iterations shown in Equation 6, we reach a result with 58-bit precision.

Shortening the estimating time. To reduce the amount of machine cycles per N-R iteration, we use the following techniques. Figure 12b shows that in an N-R iteration two data transfers between multiplier and subtracter are necessary, and that they cause the overhead of almost 30 percent of the machine cycles. We slightly tampered with the algorithm and replaced the subtraction operation with a complementing operation, which we show in Figure 12c. This operation can be realized with a small amount of hardware. The expression $3 - B * (X_i)^2$ is very close to a [one's complement of $B * (X_i)^2$] + 1.0. An error with the weight of one LSB (least significant bit) creeps in, yet it can be proven that its effect is essentially negligible. The subtraction $3 - B \Sigma (X_i)^2$ can be achieved only by inverters installed in the multiplier. This improvement reduces the overhead by 30 percent and the N-R iteration completes in 12 cycles.

Based on a pseudo solution obtained by the N-R method, the LSBs of the mantissa are recovered, and the classical restoring method of the square-root algorithm starts near the LSB. When the iterations of the restoring method complete, intermediate values are collected and a sticky bit is determined. The rounding operation yields the result required by the IEEE floating-point standard.

Figure 13a illustrates how the pencil-and-paper algorithm is actually restarted. The pseudo square root obtained by the N-R method is in the neighborhood of [pRoot – E_n, pRoot + E_n], where the real square root




```
function pencilAndPaperSquareRoot(rX);

}
This is a classic method to obtain a square root used in most IEEE conforming
floating-point arithmetics. The parameter rX lies in the interval (1.0, 2.0)
to simplify the argument. The indices are assigned for mantissa bits in the
following manner :

  0   -1   -2                -(q-1)              -N+1   N   -N-1
+---+----+---+-----------------------+---------+---+---+---+
| 1 |                                                 | R | S |
+---+----+---+-----------------------+---------+---+---+---+
    ^ binary point
}

    function searchRoot(iRoot,iResidue, k);
{
searchRoot add valid -k bit to square root solution and return result. N+1
Recursive invocations yield N+2 bits of solution.
}
        begin
        if (k = N + 1)
            then if (iResidue = 0) searchRoot := iRoot
                 else begin
                      searchRoot := iRoot + 2-N-1;
                      iResidue := newResidue(iRoot, iResidue);
                      end
        else if (root is in lower half of the interval)
            then searchRoot := searchRoot(iRoot, iResidue, k+1)
            else searchRoot := searchRoot(iRoot + 2-(k-1), iResidue, k+1);
        end;

    begin
    iRoot := 1.0;
    rRoot := searchRoot(iRoot, 0.0, 1);
    pencilAndPaperSquareRoot := round(rRoot);
    end;
```

(b)

```
function hybridSquareRoot(rX);

{   pRoot : pseudo root obtained by Newton-Raphson algorithm
    iRoot1 : a candidate intermediate value to restart pencil/paper method
    iRoot2 : a candidate intermediate value to restart pencil/paper method
    iX1 : square value of iRoot1
    iX2 : square value of iRoot2
    rRoot : root solution with ground and sticky bits
}

    function searchRoot(iRoot, iResidue, k);
{       determines one more valid bit -k of rcot and return result
}

    function newtonRaphson(rX);
{       returns pseudo solution pRoot lying in neighbourhood
        (rRoot-En, rRoot+En) of real solution rRoot (|pRoot - rRoot| ≤ En).
        An integer quantity q used in other part of the program is an integer
        such that e < 2-q-1.
}
    function clearBits(intValue, l, m);
{       clear bits l through m of intValue
}

    begin
    pRoot := newtonRaphson(rX);
    iRoot1 := clearBits(pRoot, -p, -n-1);
    iX1 := iRoot1 * iRoot1;
    iResidue1 := rX - iX1;
    if (iResidue1 = 0) then rRoot := iRoot1
    else if (iResidue1 > 0)
        then begin
             iRoot2 := iRoot1 + 2-(q-1);
             iX2 := iRoot2 * iRoot2;
             iResidue2 := rX - iX1;
             if (iResidue2 = 0) then rRoot := iRoot2
             else if (iResidue2 > 0)
                  then rRoot := searchRoot(iRoot2,iResidue2, q)
             else rRoot := searchRoot(iRoot1, iResidue1, q);
             end
        else begin
             iRoot2 := iRoot1 - 2-(q-1);
             iX2 := iRoot2 * iRoot2;
             iResidue2 := rX - iX1;
             if (residue2 = 0) then rRoot := iRoot2
             else if (residue2 < 0)
                  then rRoot := searchRoot(iRoot2, iResidue2, q)
             else rRoot := searchRoot(iRoot1, iResidue1, q);
             end;
    hybridSquareRoot := round(rRoot);
    end;
```

(c)

Figure 13. Hybrid algorithm to obtain an IEEE square root. Before starting the pencil-and-paper method (b), one must make the interval in which the real square root lies (a) the subset of the interval of convergence. The hybrid square-root algorithm used in the Gmicro/FPU (c). For a machine with a parallel multiplier, a square root can be obtained faster.

lies. The [pRoot – E_n, pRoot + E_n] neighborhood is again a subset of some interval within which the IEEE square root can be determined by the pencil-and-paper method. Figure 13b shows the Pascal-like outline of the conventional pencil-and-paper algorithm to obtain the IEEE square root. Figure 13c shows how its shortcut reconstructs the intermediate values for the pencil-and-paper method from the pseudo root obtained by the N-R algorithm. The pencil-and-paper algorithm starts from near the LSB.

Exception handling. One of the areas in which programming techniques cannot conceal the shortcomings of hardware is that of exception traps. When the function of an exception trap is realized with software, the overhead is enormous and visible. The Gmicro/FPU provides trap facilities in all operating modes, and no information is lost in a floating-point arithmetic exception, relieving programmers from any apprehensions about exception handling. The FPU

- *contains a command pipeline that does not alter the command stream.* Even though the processing is done in parallel, the sequential execution model is retained. Programmers do not have to pay attention to pipelining unless an exception trap occurs.
- *retains the information such as the instruction codes, instruction address, all operands that were used in the instruction, and exception flags.* The user can extract the flags with the floating-point-state save instructions FSAVE and FSTC. From the point at which the exception occurred, the user can deduce the cause of the problem.
- *retains the state in all pipes of the pipelines when an exception occurs.* The FPU also retains all instructions in process. Using this information, programmers can fix the problem or even restart the entire program from the point of exception after fixing some of the intermediate values.

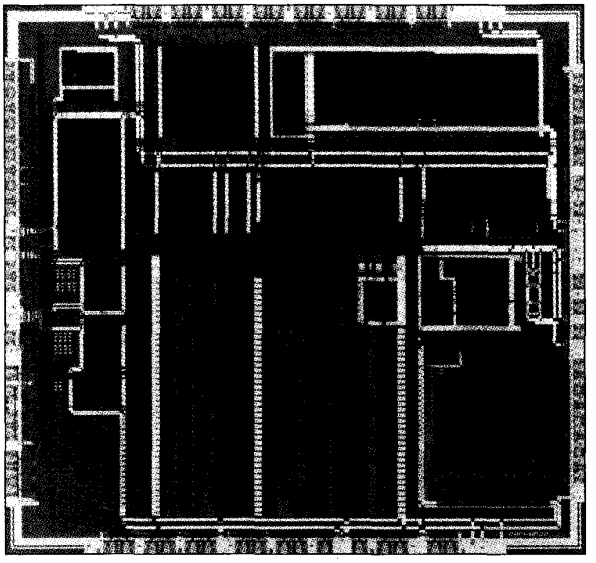


Figure 14. A photomicrograph of the Gmicro/FPU.

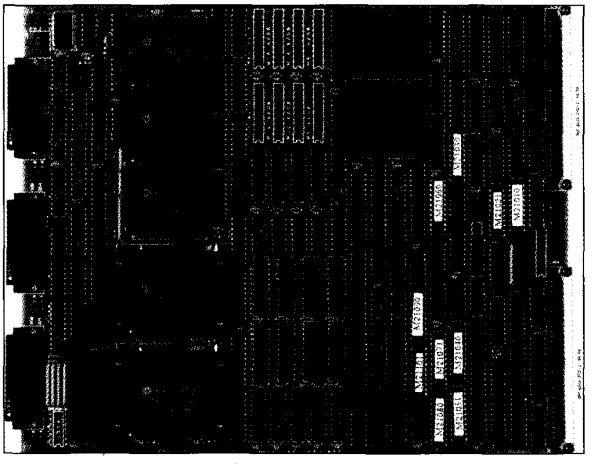


Figure 15. Gmicro/FPU evaluation boards.

Chip design. Figure 14 is a photomicrograph of the Gmicro/FPU. The FPU can be fabricated with advanced 1.0-micrometer, CMOS design technology to operate at a 20-MHz machine cycle. We implemented it in a 13.37mm × 14.05mm die, adopting 135-pin PGA packaging and integrating 540,000 transistors. The sixteen 80-bit general registers, two arithmetic units, multiplier, constant ROM, and microcode ROM are all implemented in this chip.

Table 4.
Performance figures for the Gmicro/200-Gmicro/FPU system with a 20-MHz clock rate.

Instruction execution times			
Floating-point arithmetic operation	Single	Double (microseconds)	Double-extended
Addition/ subtraction	0.5	0.5	0.5
Multiplication	0.45	0.95	0.95
Division	1.5	2.95	3.5
Square root	3.55	7.9	13.3
Sine function	6.2	8.6	8.6

Whetstone benchmark results using a C compiler object		
Variable precision	Whetstone figure (Mwips)	Code size (bytes)
Single (32 bits)	3.71	2,172
Double (64 bits)	3.32	2,466

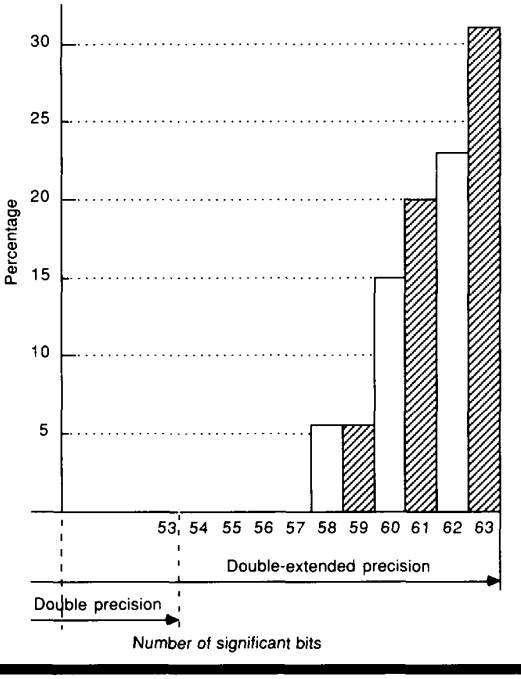


Figure 16. Precision evaluation result on elementary functions. The significant bits of the results can be seen on the random source operand. We randomly selected 500 sampling points on four representative functions.

Preliminary evaluation

We have fabricated a series of the prototype silicon for the Gmicro/FPU and have found all essential functions to be working. We evaluated the performance of the Gmicro/200 with the Gmicro/FPU system at 20 MHz and no wait states. See Figure 15 for the evaluation boards in which we embedded one Gmicro/200 and four FPUs. We used the four FPUs merely to evaluate the arbitration mechanism among multiple coprocessors.

Performance. Table 4 presents the representative instruction execution time of the Gmicro/200-Gmicro/FPU system. With single-precision format data, addition and multiplication take approximately the same amount of time to complete. The internal multiplier works on a multiplication operation in much the same way as an ALU works on an addition operation. The multiplication proceeds slightly faster than an addition because no adjustment of the multiplicand and no scaling of the result are necessary. Two factors determine the cycle time for a square-root solution: the precision of the N-R pseudo result and the operand precision of the destination operand. If we create an appropriate data path, the figures for square-root operations improve to half of what is indicated, but we do not necessarily reflect the potential vigor of the algorithm with this method. A simple restoring algorithm completes division operations.

Table 4 also gives the result of Whetstone benchmark testing on the Gmicro/200-Gmicro/FPU system using an experimental C compiler. Single-precision data executed in 3.71M Whetstone instructions per second (Mwips). The experimental compiler differs from the production compiler in the following areas: 1) we used in-line expansion for procedure calls, 2) we checked the floating-point exceptions with a floating-point trap, and 3) we used floating-data mathematical functions to suit future ANSI expansions. The production C-compiler product conforms to the current ANSI standard and yields a lower figure than the one obtained with the experimental compiler.

The static instruction count within the loops is 218. An experimental Fortran compiler generates 179 instructions for the same set of loops. Thus, we expect Fortran to generate faster code.

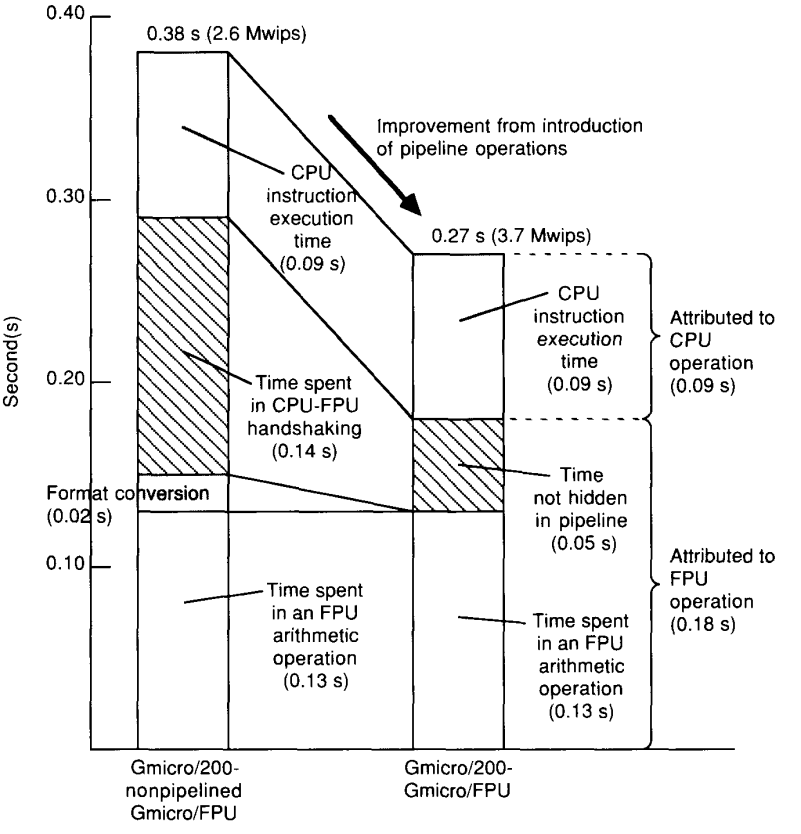


Figure 17. The effects of instruction pipelining in the Gmicro/200-Gmicro/FPU system. The figure on the left shows the Gmicro/FPU performance with pipelining suppressed.

Precision. Figure 16 displays the result of the precision evaluation on elementary functions. The elementary functions generally produce between 58 and 62 significant bits except for a singular point arising from the nature of functions and algorithm. Most libraries or floating-point units result in 53 and 58 significant bits. So the precision of the elementary functions of the Gmicro/FPU is comparatively high by more than one digit.

Instruction pipelining. Figure 17 shows the effects of using an instruction pipeline in the Gmicro/200-Gmicro/FPU system. We measured the system with Whetstone tests on C and evaluated the actual system. In addition we simulated the system with the pipeline suppressed to obtain the figures for the Gmicro/200-nonpipeline Gmicro/FPU version. When the pipeline is activated, the performance improves by 30 percent.

An interesting aspect is that the bare floating-point arithmetic operation accounts for less than half the execution time. Even in a floating-point-intensive benchmark like the Whetstone, the bare execution time of the floating-point operation is not the only factor in determining the system performance.

We have successfully developed the floating-point extension for the Gmicro VLSI CPUs, implementing architectural support for the ANSI C draft proposal and the IEEE floating-point standard in this chip. By exploiting the high performance of the recent multiplier circuitry, we propose improvements on the Cordic algorithm and the IEEE square-root algorithm for this chip. New MPU-FPU handshaking protocols and an instruction pipeline mechanism contribute to the reduction of communication overhead among the FPU, memory, and CPU. Preliminary evaluations show a good combination of precision and performance. The elementary function performance indicates higher precision by one digit than is typical in other implementations. The Whetstone performance measures at 3.71M Whetstone instructions per second.

References

1. K. Sakamura, "The TRON Project," *IEEE Micro*, Apr. 1987, pp. 8-14.
2. K. Sakamura, "Architecture of the TRON VLSI CPU," *IEEE Micro*, Apr. 1987, pp. 17-32.
3. H. Inayoshi et al., "Realization of the Gmicro/200," *IEEE Micro*, Apr. 1988, pp. 12-21.
4. C. Farnum, "Compiler Support for Floating-Point Computation," *Software-Practice and Experience*, Vol. 18, No. 7, pp. 701-709.
5. K. Sakamura, *Introduction to ITRON: Concepts and Implementations*, Iwanami, Tokyo, 1988, pp. 5.
6. *IEEE Standard 754-1985 for Binary Floating-Point Arithmetic*, IEEE Computer Society Press, Los Alamitos, Calif., 1985.
7. *Draft Proposed American National Standard for Information Systems—Programming Language C*, Doc. No. X3J11/88-90, American National Standards Institute, 1988.
8. T.C. Chen, "Automatic Computation of Exponentials, Logarithms, Ratios and Square Roots," *IBM J. Research and Development*, Vol. 16, July 1972, pp. 380-388.
9. J.E. Volder, "The CORDIC Trigonometric Computing Technique," *IRE Trans. Electronic Computing*, Vol. EC-8, Sept. 1959, pp. 330-334.
10. J.S. Walther, "A Unified Algorithm for Elementary Functions," Spring Joint Computing Conf., *AFIPS Conf. Proc.*, Vol. 38, 1971, pp. 389-385.
11. H. Kida et al., "A Floating-Point Processing Unit for the Gmicro CPU," *TRON Project 1988, Open-Architecture Computer Systems*, Springer-Verlag, Tokyo, 1988, pp. 301-316.

Additional Reading

J.T. Coonen, "An Implementation Guide to a Proposed Standard for Floating Point Arithmetic," *Computer*, Jan. 1980, pp. 68-79.

H.J. Curnow, "A Synthetic Benchmark," *The Computer Journal*, Vol. 19, No.1, Jan. 1976, pp. 43-49.

G.L. Haviland, "A CORDIC Arithmetic Processor Chip," *IEEE Trans. Computers*, Vol. C-29, Feb. 1980, pp. 68-79.

R.K. Richards, "Arithmetic Operations in Digital Computers," D. Van Nostrand Company, Inc., Princeton, N. J., 1955, pp. 291-295.

Shumpei Kawasaki

Mitsuru Watabe

Shigeki Morinaga

Shumpei Kawasaki, an engineer at Hitachi's Semiconductor Design and Development Center, currently holds responsibility for the design and development of its 32-bit floating-point chip. His interests include high-level-language compilers, instruction architecture, and Japanese script management.

Kawasaki received the BA in mathematics from Knox College in Galesburg, Illinois, and an MS in computer science from the University of Illinois at Urbana-Champaign.

Mitsuru Watabe is a researcher at the Hitachi Research Laboratory. He has been working with VLSI architecture for machinery control, and his interests include electromachinery control by microcomputers. Watabe received BE and ME degrees in electrical engineering from Musashi Institute of Technology, Setagaya, Tokyo.

Shigeki Morinaga, a senior researcher in the Hitachi Research Laboratory, has been engaged in the research and development of control LSI chips. He received BE and ME degrees in electrical engineering from Kyushu University, Fukuoka City, Fukuoka.

Questions about this article can be addressed to Shumpei Kawasaki, Semiconductor Design and Development Center, Microcomputer Engineering Dept., Hitachi, Ltd., 20-1 Josuihon-cho 5 Chome, Kodaira-shi, Tokyo, Japan 187.

PROGRAMMING LANGUAGE CONCEPTS

Carlo Ghezzi
Politecnico di Milano

Mehdi Jazayeri
Synapse Computer Corporation

John Wiley & Sons, Inc.
New York Chichester Brisbane Toronto Singapore

To Anny ar

Library of Congress Cataloging in Publication Data:

Ghezzi, Carlo.
Programming language concepts.

Bibliography: p.
Includes index.
1. Programming languages (Electronic computers)
I. Jazayeri, Mehdi. II. Title.

QA76.7.G48 001.64'24 81-16032
ISBN 0-471-08755-6 AACR2

Printed in the United States of America

10 9 8 7 6 5 4 3 2 1

CHAPTER FIVE

CONTROL STRUCTURES

This chapter is devoted to a detailed analysis of control structures—the mechanisms by which programmers can specify the flow of execution among the components of a program. In Section 2.3, we distinguished between unit-level control structures and statement-level control structures. For simplicity, we will start our analysis with the latter category. Unit-level control structures and their implementation models will be studied in Section 5.2.

5.1 STATEMENT-LEVEL CONTROL STRUCTURES

There are three kinds of statement-level control structures: sequencing, selection, and repetition. We analyze each of these separately, in Sections 5.1.1 through 5.1.3.

Statement-level control structures strongly contribute to the readability and maintainability of programs. In fact, for the intellectual manageability of programming it is crucial that instructions be connected according to sufficiently simple, natural, and well-understood schemes.

5.1.1 Sequencing

Sequencing is the simplest structuring mechanism available in programming languages. It is used to indicate that the execution of a statement (B) must follow the execution of another statement (A). This is usually written as

$A;B$

where ";" (which can be read as "and then") denotes the sequencing control operator. Languages that adopt a line-oriented format (e.g., FORTRAN) use the end of the line implicitly to separate instructions and superimpose a sequencing mechanism among

them. It is possible to group together statements of a sequence to form a unique *compound* statement. Several languages (e.g., ALGOL 60 and Pascal) use the bracketing keywords **begin**, **end** for this purpose. For example: **begin** *A*; *B*; . . . **end**.

5.1.2 Selection

Selection control structures allow the programmer to specify that a choice is to be made among a certain number of possible alternative statements.

The logical IF statement of FORTRAN is an example of a selection statement that specifies the execution of a statement according to a boolean expression. For example, the FORTRAN statement

```
IF (I.GT.0) I= I-1
```

decreases the value of *I* by one if *I* is positive.

More general and powerful is the **if** statement of ALGOL-like languages, in which the presence of an **else** branch allows the programmer to choose between two control paths as a consequence of a test. Here, unlike in FORTRAN, a branch can be any statement (e.g., a compound statement). For example, the program fragment

```
if i = 0
  then i := j
  else begin i := i + 1;
             j := j - 1
       end
```

sets i to the value of j if $i=0$; otherwise, it sets i to $i+1$ and j to $j-1$. If the keywords **begin**, **end** were omitted, $j:=j-1$ would have been considered as a statement following the selection, and thus also executed in the case $i = 0$.

The selection construct of ALGOL 60 raises a well-known ambiguity problem. In the example

```
if x>0 then if x<10 then x:= 0 else x:= 1000
```

it is not clear whether the **else** branch is part of the innermost conditional (**if** $x<10$ . . .) or part of the outermost conditional (**if** $x>0$ . . .). The execution of the above statement with $x=15$ would assign 1000 to x under one interpretation, and leave it unchanged under the other. To eliminate ambiguity, the ALGOL 60 syntax requires an unconditional statement in the **then** branch of an **if** statement. Thus, the above fragment must be replaced either by

```
(i) if x>0 then begin if x<10 then x:= 0 else x:= 1000 end
```

or by

```
(ii) if x>0 th
```

according to

The san
else branch t
would be unr
language defi
ditional struc
careful inden
desired interp

A syntac
uses the keyv
statements is
This makes i
statement. Th

```
if i=0
  then i:= j
  else i:= i
       j:= j
fi
```

and

```
if x>0 then i
```

or

```
if x>0 then i
```

according to
uses the keyv

Both AL
alternatives c
ward progran

```
if a
  then S1
  else if b
          then
          else

          fi
fi
```

On Tuning the Microarchitecture of an HPS Implementation of the VAX

James E. Wilson, Steve Melvin, Michael Shebanow, Wen-mei Hwu, and Yale N. Patt

Computer Science Division
University of California, Berkeley
Berkeley, CA 94720

Abstract

The HPS Microarchitecture has been developed as an execution model for implementing various architectures at very high performance. A considerable amount of effort has gone into the use of HPS as a microarchitecture for the VAX. In this paper, we describe our first full simulation of the microVAX subset, and report the results of varying (i.e. tuning) certain important parameters.

1 Introduction

HPS (High Performance Substrate) is a microarchitecture for implementing high performance computing engines. It exploits concurrency by using a restriction of classical fine granularity data flow [6]. This is a part of the Aquarius project, whose goal is to obtain enormous improvements in computer performance, in part by exploiting concurrency at all levels of transformation. HPS represents our approach to dealing with concurrency at the microarchitecture level.

This paper builds upon the work of two previous papers. The first paper [4] covered the initial design for an HPS implementation of the VAX. It described the functional units necessary to implement the VAX, and provided some encouraging preliminary results. The second paper [7] explained details of how to generate data flow nodes from a sequential VAX instruction stream. This paper covers the implementation of a simulator for the microVAX subset. Results obtained from running a set of benchmarks on the simulator are presented.

Section 2 of this paper describes the HPS VAX specification, including some discussion of the parameters to be tuned. Section 3 describes the simulator model. Section 4 reports simulation results obtained by tuning certain parameters, and provides an analysis of these results. Finally, section 5 offers some concluding remarks.

2 HPS/VAX Implementation

2.1 HPS Execution Model

The HPS execution model is a restriction on classical fine granularity data flow, hence we are calling it "restricted data flow".

However, there are substantial differences between our model and the classical fine granularity data flow engines of Dennis[3] and Arvind[2]. The most important difference is that our data flow oriented microarchitecture is implementing a conventional control flow oriented ISP architecture. We define the "active window" as the set of ISP instructions whose corresponding data flow nodes are currently part of the data flow graph which is resident in the microengine. Consequently, in our model, only a small subset of the entire program is present in our microengine at any instant of time. As the active window moves through the dynamic instruction stream, HPS executes the entire program.

An abstract view of HPS is shown in Figure 1. The static instruction stream is fetched, with branch prediction, to create the dynamic instruction stream. Branch prediction is necessary to prevent stalls due to branches. Incorrect predictions are handled by the checkpoint/repair mechanism.

The decoder takes instructions from the dynamic instruction stream and generates data flow nodes for the instructions. Only instructions within the active window will have nodes active in the machine, unlike classical data flow in which the entire program must be in the machine. This is why we refer to HPS as restricted data flow. Parallelism that exists within the active window will be fully exploited by the data flow microengine.

The merger takes nodes generated by the decoder and merges them into the entire data flow graph for the active window. A generalized version of the Tomasulo algorithm[8] is used to resolve all data dependencies existing between the new nodes and the nodes in the machine. The merger uses information stored in

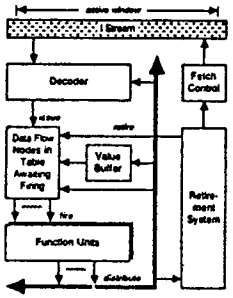


Figure 1. HPS Overview

the Register Alias Table (RAT) and the Scratch Pad Alias Table (SPAT) to help resolve these dependencies. The merged nodes are then stored into the node tables.

The scheduler searches the node tables for nodes that are ready to fire. A node is ready to fire when all of its operands are ready. The oldest node in each table with all of its operands ready is sent to the function units for execution. A typical implementation could have five function units: 2 fixed point, 1 memory, 1 branch, and 1 floating point. This scheme transparently handles multiple function units and out of order execution.

After execution, the results of each node must be distributed. To do this, each function unit gates its result to its own distribution bus, which is connected to all the node tables and alias tables. A value is put on the bus along with a tag, any node or alias table entry that has a matching tag will load the value and set its ready bit.

Finally, all instructions go through the retirement stage. Instructions are sequentially retired, after all of their nodes have been executed. This enforces the sequential semantics of the target architecture, allowing the HPS implementation to handle exceptions and interrupts identically to traditional implementations (i.e., precise interrupts[1]). The retirement stage interacts with the checkpoint/repair mechanism to achieve this.

2.2 Functional Elements

This section covers the functional elements of the HPS VAX implementation.

2.2.1 Decoder

The decoder takes an instruction and generates the data flow nodes necessary to execute that instruction. This mainly involves inserting operand values into known templates of nodes. The operand specifiers are decoded first, the nodes generated depend only on the addressing mode. Next, the nodes for the opcode are generated. For most instructions, this just involves taking the known template of nodes, and inserting operand values into the appropriate locations.

For some instructions however, the data flow nodes needed depend on the values of the operands. For example, the PUSHR instruction pushes registers onto the stack as indicated by a register mask. Since a node must be generated only for each register to be pushed, the nodes cannot be generated until this value is known. If this mask is not an instruction stream literal, then the decoder will stall until this value becomes known. This is referred to as a node generation stall.

Another example that causes stalls are CALLS and RET instructions. Since the next instruction address will come from the stack, the decoder may have to stall if this value is not known by the time it has finished decoding this instruction. This is referred to as a branch destination stall.

2.2.2 Node Cache

The performance required from the decoder can be reduced by storing previously decoded instructions in a node cache. A hit in the node cache means an effective decode time of 1 cycle. Unfortunately, this does not solve all of the problems, because not all instructions can be cached. In general, an instruction that causes a stall can not be cached. For example, the PUSHR instruction can not be cached if the register mask is not an instruction stream literal. In this case, the nodes needed for two different executions of this instruction will be different if the mask changes. We have measured the frequency of occurrence of these stall conditions[7]. Fortunately, in the case of the PUSHR instruction, in all benchmarks measured, the register mask was always an instruction stream literal.

2.2.3 Value Buffer

Each node in the machine has a unique tag associated with it. These tags are used to uniquely identify every value in the machine, and can be thought of as identifying the arcs in a data flow graph. These tags allow a node to use the value of a previous node by referring to the tag of the previous node. All values in the machine are stored in the value buffer (VB). The tag is an index into the VB. Note, the tags wrap around at the end of the VB. The size of the VB is a function of the window size. For a window size of 8 instructions, an eight bit tag is sufficient.

A special area (the low 32 locations) of the VB is reserved for the permanent values of the dynamic registers. Since the VB is a circular buffer, all values are eventually overwritten. To prevent the loss of register contents, they are stored in this area. When an instruction retires, registers modified by the instruction must have their values copied to this area.

2.2.4 Register Alias Table

The register alias table (RAT) manages all data dependencies involving the registers. For each register, the RAT contains a ready bit and a tag. If its ready bit is set, then the register's value is in the value buffer. If its ready bit is clear, then its value is not yet known.

The machine registers are divided into two categories, static registers and dynamic registers. Static registers are those which change infrequently or must be known to the decoder, while dynamic registers are those which are changed frequently and don't have to be known by the decoder. For example, most of the Program Status Longword (PSL) must be known by the decoder at all times, and hence is implemented as a static register, whereas the general purpose registers are implemented as dynamic registers.

Dynamic registers are represented by entries in the RAT. The RAT aliases a tag to each dynamic register, so that these registers can be referenced even if their value is not currently known. Static registers, on the other hand, force the decoder to stall when they are written to. The trade off is that much more hardware is needed for dynamic registers than static registers.

The RAT has entries for the 16 general purpose registers, the four condition codes, and the five architecturally defined stack pointers. Even though the condition codes are part of the PSL, they need to be implemented as dynamic registers as most instructions modify them. Making the stack pointers dynamic registers prevents the Change Mode (system call) instructions from generating stalls.

2.2.5 Scratch Pad Alias Table

The Scratch Pad Alias Table (SPAT) is needed by the merger to correctly handle instructions that take multiple cycles to merge. For every node generated by the decoder for the current instruction, there is a slot in the SPAT that corresponds to it. Each entry in the SPAT is a single bit that indicates whether or not the value of the corresponding node is valid. Note that this means the SPAT must also be loaded from the various distribution buses. When merging a node that depends on a previous node (within the same instruction), the merger checks the SPAT to see whether or not the operand ready bit should be set.

Consider the case of an instruction which takes two cycles

to merge. On the second merge cycle, there will be nodes that refer to nodes merged during the first merge cycle. Usually, these nodes will not have been executed, but this is not always true. There could have been a stall due to a full active instruction window, for instance. During this stall, nodes from the first merge cycle could have executed before the second merge cycle occurred. If the rest of the nodes were subsequently merged with their operand ready bits clear, they would never execute as they would be waiting for values which had already been distributed. The SPAT solves this problem by always indicating which nodes of the current instruction have been executed.

2.2.6 Merger

The merger takes nodes from the decoder and resolves all dependencies between the new nodes and the data flow graph for the active window. This involves assigning unique tags to each node, and making sure that each internodal reference uses these tags. Each register read is replaced by the tag and ready bit for that register from the RAT. Each register write results in the RAT tag for the appropriate register being set to the tag of the node modifying it. Also, the ready bit for the register is cleared. This operation is done at the end of the cycle, so that all nodes merged during the same cycle will see the same tag for each register. The condition codes are handled in the same way, each node that modifies a condition code updates the RAT entry for that condition code.

2.2.7 Node Tables

After the merge process is done, the nodes are in the node tables. The node tables hold nodes while they await execution. The node table can be represented as a circular buffer, with new nodes being inserted at the tail. Each node table entry has a valid bit to indicate whether or not it contains a node. The nodes themselves have two ready bits, one for each operand. Every cycle, the node closest to the head that has all operands ready will be sent to a function unit for execution. This entry can be determined by a simple priority circuit.

There is a node table for each function unit. This simplifies the scheduling of nodes, as at most one node per cycle needs to be read from each node table. This, however, does create the problem of how to store nodes in the node tables for which identical function units exist.

2.2.8 Function Units

There are four types of function units in the simulator: Fixed Point, Floating Point, Memory and Branch. (Floating Point functions have not been implemented.) The Fixed Point function units (identical if more than one) have 24 operations defined. These include the expected arithmetic operations, special operations for doing double word (slc) multiplies and divides, and some special operations for manipulating the condition code fields of values. (It should be pointed out that each "value" in the machine is actually a 32 bit value plus a four bit condition code.) The memory function units have 8 operations defined. Read and write are by far the most commonly used ones; the rest are needed for virtual memory and operating system support. The branch function units have 12 operations defined which cover all condition code combinations used by VAX branches.

2.2.9 Distribution Bus

After a node is executed, its value must be distributed on the distribution bus to all locations that may depend on it. This includes the node tables, the RAT and the decoder. At each tag location, the tag on the distribution bus is compared to the stored tag. If they match, then the ready bit is set. At the same time, the value on the bus will be written into the value buffer.

Note that there is a distribution bus for each function unit. This does not cause contention problems, however, since each tag knows which distribution bus its result will appear on, and the tag comparisons only have to be done against that bus.

2.2.10 Checkpoint/Repair

The Checkpoint/Repair mechanism is used to ensure that precise interrupts will occur, and to maintain the sequential semantics of the VAX architecture. There are two major reasons for this mechanism: branch prediction and exceptions. In the case of an incorrect branch prediction, the effects of all instructions after the branch must be undone, and the correct execution path must be followed. For exceptions, the effects of all instructions after the one that caused the exception must be undone, and an exception handler must be executed. The simulator currently treats these situations identically.

When an exception or incorrect branch prediction is recognized, a repair is initiated. Repair involves restoring the state to a point in the instruction stream corresponding to the instruction that caused the problem, and invalidating all nodes merged after this instruction. Execution then resumes at the correct point, either an interrupt handler in the case of exceptions, or the correct branch direction in the case of branches.

3 Simulation Model

The simulator is written in about 24,000 lines of C. It consists of the microVAX subset of the VAX architecture, plus an additional 8 VAX instructions for a total of 193 VAX instructions implemented. There are 132 instructions unimplemented. The unimplemented instructions fall into three categories: floating point, decimal, and character string. Two character string instructions are implemented, MOVC3 and MOVC5; they are used for block transfers. The memory system is implemented as follows. Instruction fetches each take 1 cycle, assuming a 100% prefetch buffer hit rate. We also assume a 100% TLB hit rate and a 100% cache hit rate for all data accesses. Therefore, all data accesses take the same number of cycles, as specified by the memory function unit latency.

3.1 Simulator Flow Chart

The simulator uses a two phase clocking scheme. It assumes that all data structures (latches, registers, etc.) can be accessed twice per cycle, once in the first phase and once in the second phase.

The first phase includes the scheduling and merging stages of the pipeline. The scheduler must read each node table, presumably near the beginning, to send a node to the execution units. The merger must read and update the contents of the register and operand stack alias tables. After resolving data dependencies, the merger writes nodes to the rear of the node tables.

The checkpoint/repair mechanism also works during the first phase. Checkpointing the state of the alias tables is a simple operation that occurs in parallel with the merger alias table accesses. Repair is more involved. When a repair is initiated, no scheduling or merging is allowed to occur that cycle. The retirement mechanism also occurs during the first phase as it is closely related to the checkpoint/repair mechanism.

The second phase includes decoding and distribution. The decoder generates nodes for the merger, but does not otherwise interact with the main datapath. The distribution stage will

beachE	string search
beachF	linked list insertion
beachH	bit test and set
hanoi3	Towers of Hanoi with 3 disks
linpack	the daxpy subroutine from linpack
dhrystr	Dhrystone, with no registers

Figure 2. Benchmarks

update the contents of the node tables and alias tables, and hence must occur in the opposite phase from both the merging and scheduling phases.

The execution units in this scheme take as long as necessary. Operations which are assumed to execute in a single cycle will be scheduled and distributed in the same cycle.

3.2 Benchmarks

The HPSVAX simulator takes as input a UNIX executable program, and executes the program. The simulator provides a bare VAX execution environment. For example, virtual memory won't work unless you set up the page tables yourself. Also, system calls will not work unless you provide your own code for the system calls. As a result, the simulator can execute any compiled C program that does not have any system calls. This necessarily excludes I/O, but does not exclude most of the library routines. The benchmarks used for this paper are shown in Figure 2.

The initial set of benchmarks are all relatively small programs. As a result, they do not provide a good test of many HPS features, such as the node cache. Future work will include measuring the performance of larger benchmarks on the simulator.

BeachE is a string search benchmark. This program spends most of its time in a simple 4 instruction loop. Since this loop can be predicted well, this benchmark is near 1 cycle/instruction for most simulator configurations. Also, six VAX instructions account for 86% of all instructions executed.

BeachF is a bit test and set benchmark. In this benchmark 12 instructions account for 90% of those executed.

BenchH is a linked list insertion benchmark. Two instructions, MOVL and CLRL, comprise 44% of those executed.

The Hanoi3 benchmark is a recursive program that solves the Towers of Hanoi problem with three disks. Six instructions account for more than 98% of those executed. Also, almost one fourth of the instructions executed are CALLS or RET instructions. This is far from the norm for C language programs.

The linpack benchmark is the daxpy subroutine from the linpack benchmark set. This is an integer precision version of the subroutine. Three instructions account for more than 98% of those executed.

The Dhrystone benchmark is a C version of the ADA benchmark written by Reinhold P. Weicker. This benchmark was developed by examining the dynamic instruction set usage of typical programs, and then written to try to match those statistics. Eighteen instructions account for 90% of those executed.

3.3 Parameters

Figure 3 shows a sample configuration file for the simulator. The window size indicates how many instructions are allowed to be active at a time. When there are this many instructions in the active window, the decoder will stall until the oldest one retires. The larger the window, the more entries are needed for the node tables. In this case, the node tables have 64 entries each. The value buffer has 64 entries in each of its four banks: LIT, FIX, MEM, and REG. Note that the branch unit doesn't have a bank because it does not produce results.

The merger bandwidth is assumed to be the same as the number of functions units, except for memory nodes and literals. There is never more than one memory function unit, i.e. the memory system is assumed to be able to initiate only one access at a time. The literals do not have an associated function unit. They are used to insert 32 bit quantities into the datapath (such as static registers, absolute addresses, etc.) and are directly distributed after being merged. The floating point function units are not implemented since the microVAX subset does not include floating point instructions. We only have one branch unit since, in the absence of string instruction optimization, VAX instructions contain at most one branch node each.

For each instruction, the register/memory write buffer indicates which register/memory locations are modified by the instruction. This information is used by the retirement system to move register values to the permanent area of the VB, and to make memory writes permanent by sending them to the main memory. The Memory alias table is an associative buffer that holds all memory writes until retirement time.

Two parameters, the register and memory write retirement rates, indicate how many writes, of each type, can be retired per cycle. We are developing schemes to make retirement transparent to the rest of the machine. This is reflected in the large default values for retirement rates that we use in the simulator.

The final group of parameters specify the node cache. The node cache size indicates the number of entries, where each entry consists of all of the nodes for an instruction. The set size indicates the associativity of the node cache. The Hit Decode Penalty indicates how many cycles it takes the decoder to generate nodes for an instruction when there is a node cache hit. The Miss Decode Penalty indicates the number of cycles for node generation when there is a node cache miss.

Not shown in the figure are the latency figures for the function units. The Fixed Point and Branch function units are usually assumed to take one cycle. The memory unit default latency is two cycles.

The values shown in Figure 3 comprise the default configuration for all the simulations below. When a set of parameters are varied, the rest of the parameters are assumed to be these default values.

8	Window Size
64	Fix Point Node Table Bank Size
64	Memory Node Table Bank Size
64	Branch Node Table Bank Size
64	VB Bank Size
4	Number of Fixed point FU
2	Memory Merging Bandwidth
0	Number of Floating Point FU
1	Number of Branch FU
4	Literal Merging Rate
64	Register Write Buffer Size
64	Memory Write Buffer Size
64	Memory Alias Table Size
1024	Node Cache Size
4	Node Cache Set Size
1	Node Cache Miss Decode Penalty
1	Node Cache Hit Decode Penalty
16	Register Write Retire Rate
16	Memory Write Retire Rate

Figure 3. Standard Configuration File

4 Measurements and Analysis

This section describes the tuning experiments that we have performed with the simulator. Many parameters in the simulator can be varied. We have chosen to tune the window size, number of function units, memory latency, function unit latency, and decode delay, because of the likelihood of these parameters to influence performance.

4.1 Window Size

The active window size strongly dictates the amount of hardware in the machine. As the window size increases, the memory alias table, all of the node tables, and also the value buffer must increase in size to hold the extra nodes. The window size also has an effect on the parallelism that can be extracted from the instruction stream. The larger the window size, the more nodes are in the machine and the greater the chance that local parallelism can be exploited.

Figure 4 shows the effects of changing the window size. As expected, performance increases as window size increases. Note that for the benchmarks tested, there is no gain from increasing the window size from 12 to 16. There is however a large increase in going from 4 to 8, so the window size should be at least 8. Since a window size of 8 yields most of the performance possible, for these benchmarks, it is a good benefits/cost choice for this implementation. The hardware needed is a node table of size 64 and a value buffer of 256 entries. Both of these values are reasonable for hardware implementations.

4.2 Number of Function Units

The number of fixed point function units is varied also. The fixed point units are assumed to be all identical. Much of the work is done by these function units, such as address generation, register increment/decrement, etc. Therefore it is useful to have as many as possible. However, multiple function units require more logic in terms of the function units themselves, additional distribution buses, and, if we wish to effectively load balance the multiple function units, additional scheduling hardware. This would argue for keeping the number of function units as low as possible.

Figure 5 shows the effects of multiple fixed point function units upon performance. There is a large performance increase from 1 to 2 units, a moderate increase from 2 to 4, and very little increase afterwards. The flatness of the graph after four function units is due in large part to the fact that our implementation merges one instruction per cycle. When so limited, there is not enough work to keep 8 function units busy. Also, since we assumed a naive method of scheduling fixed point nodes, which makes no attempt to balance the load, we can not effectively utilize the additional function units. With a substantial increase in decoding, merging, and scheduling hardware, we could obtain improved results for large numbers of function units. For this implementation however, four fixed point function units offers a good balance between performance and cost.

4.3 Memory Latency

Memory latency is another important parameter affecting performance. (See Figure 6.) HPS, with its out of order execution, should not be drastically affected by memory system delays, provided that memory contention is not caused by multiple active memory nodes in various stages of execution. In fact, we have assumed that memory accesses are perfectly pipelined, so that one memory access can be started every cycle no matter what the latency is. With this assumption, the performance of HPS shows very little effect from the increase in the memory latency. This demonstrates that HPS can effectively overlap operations, and execute operations out of order to reduce the effect of the memory latency, given this caveat about contention. We should also point out that although the concept of perfect pipelining of the memory accesses may appear at first to be unrealistic, Swensen[9] has suggested a circuit for facilitating pipelined memory accesses. Also, a sufficient degree of interleaving in the main memory would effectively produce the benefits of pipelining. How much interleaving would be required to accommodate this is still an open question.

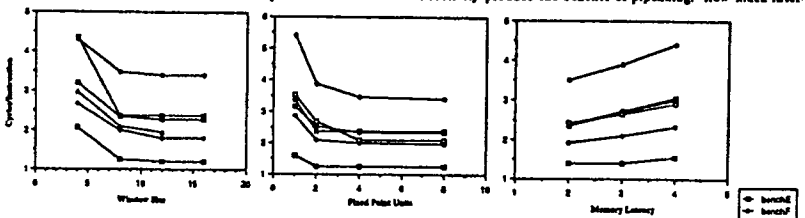


Figure 4. EFFECT OF WINDOW SIZE

Figure 5. EFFECT OF NUMBER OF FIXED POINT UNITS

Figure 6. EFFECT OF MEMORY LATENCY

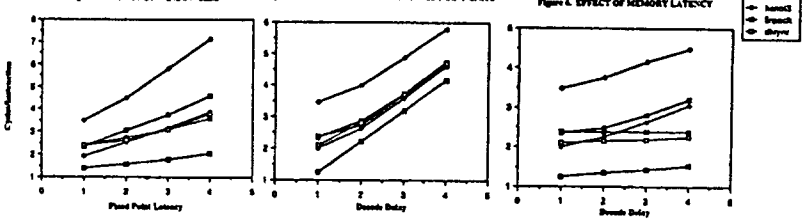


Figure 7. EFFECT OF FIXED POINT LATENCY

Figure 8. EFFECT OF DECODE DELAY, NO NODE CACHE

Figure 9. EFFECT OF DECODE DELAY WITH NODE CACHE

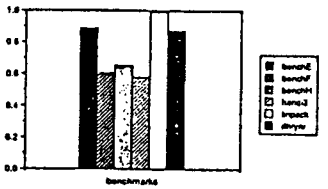


Figure 10. NODE CACHE HIT RATE

4.4 Function Unit Latency

Fixed point function unit latency is also an issue. Again, the function units are assumed to be perfectly pipelined so that each fixed point function unit can start one alu operation every cycle. This is not at all difficult to implement, and in fact can be a big win since it potentially allows dramatic reduction of the cycle time. Figure 7 shows the effects of changing the function unit latency. For four of the benchmarks, the performance decreases by roughly one half when the fixed point function unit latency is increased from 1 to 4 cycles. For the other two, the decrease is less than one half. This is due to the fact that HPS is able to effectively exploit concurrency by overlapping operations, thereby reducing the performance penalty due to the increase in the function unit latency.

4.5 Node Cache

Since the decoder is one of the most complicated parts of a VAX implementation, it is important to know the effects of decode delays on performance. The node cache handles much of this problem by caching instructions in decoded form. Our initial benchmarks are small enough to fit into a reasonably sized node cache. Hence the node cache hit rate is dominated by the percent of the instructions executed that are actually cacheable. This will change when we expand our set of benchmarks.

Figure 8 shows the effect of varying the decode delay from 1 to 4 cycles in the absence of a node cache. The decode delay is defined here as the number of cycles until the first node is generated, with all nodes then being available at the same time. With a decode delay of four cycles, the best possible performance is four cycles per instruction. So, predictably, the performance decreases as the decode delays increase. Note also, that as the decode delays increase, the performance tends to get closer to the theoretical maximum, as the HPS microengine is better able to overlap the decode and execute stages of each instruction.

Figure 9 shows the effect of varying the decode delay when a node cache is present. We note that most of the benchmarks show very little effect on performance as most of their instructions can be cached. The hanoi benchmark, for example, has a noticeable performance decrease. This is because hanoi executes many CALLS and RET instructions which can not be cached.

Figure 10 shows the node cache hit rates for the benchmarks. Linpack has a 99% hit rate which is why it is very insensitive to the decode delays in the presence of a node cache. The worst is hanoi3, at 58%. As mentioned above, the node cache at present is overly pessimistic in deciding when it can cache an instruction, so we expect to be able to improve on these results.

5 Conclusions

HPS offers the potential of a high performance implementation for traditional architectures, by providing high bandwidth, few stalls, multiple function units and out of order execution. It is relatively insensitive to parameters such as memory latency due to the out of order execution of nodes. One of the more complicated parts of a VAX implementation, the decoder, has only a moderate effect on the performance as many instructions can be cached in their decoded form (assuming that the node cache is large enough to get a good hit rate.)

The results suggest that a reasonable implementation, as defined by the standard configuration file, could execute at the rate of 2 cycles per VAX instruction. This compares very favorably with existing implementations which all take over 6 cycles per instruction. We must be quick to point out, however, that this simple comparison does not take into account some of the real world problems of computers (cache misses, TLB misses, etc.), and also suffers from the absence of real world substantial benchmarks. On the other hand, it does not fully exploit the capabilities of HPS (unnecessary decoder stalls, etc.). In summary, this paper shows that there are performance benefits to be gained by using the HPS concepts, and that future refinements are warranted.

6 Acknowledgements

The authors wish to acknowledge the Digital Equipment Corporation and the NCR Corporation for their generous support of our research. We also would like to acknowledge two other members of our research group, Chien Chen and Jiajuin Wei, who have made contributions to this project. Part of this work was sponsored by the Defense Advanced Research Projects Agency (DoD), Arpa Order No. 4871, monitored by Space and Naval Warfare Systems Command under Contract No. N00039-84-C-0089. Finally, we also acknowledge that part of this work was supported under a NSF Graduate Fellowship.

References

[1] Anderson, D. W., F. J. Sparacio, and R. M. Tomasulo, "The IBM System/360 Model 91: Machine Philosophy and Instruction Handling," *IBM J. of R & D*, vol. 11, no. 1, 1967.

[2] Arvind and K. P. Gostelow, "A New Interpreter for Dataflow and Its Implications for Computer Architecture," Department of Information and Computer Science, University of California, Irvine, Tech Report 72, October 1975.

[3] Dennis, J. B., and D. P. Misunas, "A Preliminary Architecture for a Basic Data Flow Processor," *Proceedings of the Second International Symposium on Computer Architecture*, 1975, pp. 126-132.

[4] Hwu, W., S. Melvin, M.C. Shebanow, C. Chen, J. Wei, and Y.N.Patt, "An HPS Implementation of the VAX: Initial Design and Analysis," *Proceedings of the 19th Annual Hawaii International Conference on System Sciences*, 1986.

[5] Hwu, Wen-mei, and Yale N. Patt, "Checkpoint Repair for High Performance Out-of-order Execution Machines," *Proc. 14th Int. Symp. on Comp. Arch.*, June 2-5, 1987, Pittsburgh, Pa.

[6] Patt, Y.N., Wen-mei Hwu, Mike Shebanow, "HPS, a New Microarchitecture: Rationale and Introduction," *Proc. of the 18th Microprogramming Workshop*, Asilomar, California, Dec., 1985.

[7] Patt, Y. N., S. Melvin, W. Hwu, M. C. Shebanow, C. Chen, J. Wei, "Run-Time Generation of HPS Microinstructions From a VAX Instruction Stream," *Proceedings to the 19th Annual Workshop on Microprogramming*, October 15-17, 1986, New York, New York.

[8] Tomasulo, R. M., "An Efficient Algorithm for Exploiting Multiple Arithmetic Units," *IBM J. of R & D*, vol. 11, no. 1267, pp 25 - 33.

[9] Swensen, John, "High-Bandwidth/Low-Latency Temporary Storage for Supercomputers," *PhD Thesis*, U.C. Berkeley, Nov., 1987.